

OpenStep Development Tools

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.



SunSoft
A Sun Microsystems, Inc. Business

© 1996 Sun Microsystems, Inc.

2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. All rights reserved.

Portions Copyright 1995 NeXT Computer, Inc. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® system, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and from the Berkeley 4.3 BSD system, licensed from the University of California. Third-party font software, including font technology in this product, is protected by copyright and licensed from Sun's suppliers. This product incorporates technology licensed from Object Design, Inc.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

Sun, Sun Microsystems, the Sun logo, SunSoft, the SunSoft logo, Solaris, SunOS, and OpenWindows are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. OPEN LOOK is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. Object Design is a trademark and the Object Design logo is a registered trademark of Object Design, Inc. OpenStep, NeXT, the NeXT logo, NEXTSTEP, the NEXTSTEP logo, Application Kit, Foundation Kit, Project Builder, and Workspace Manager are trademarks of NeXT Computer, Inc. Unicode is a trademark of Unicode, Inc. VT100 is a trademark of Digital Equipment Corporation. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. SPARCcenter, SPARCcluster, SPARCCompiler, SPARCdesign, SPARC811, SPARCengine, SPARCprinter, SPARCserver, SPARCstation, SPARCstorage, SPARCworks, microSPARC-11, and UltraSPARC are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of X Consortium, Inc.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



Adobe PostScript

Contents

1. Introduction	1-1
Putting Together an OpenStep Application	1-2
Designing Your Application	1-2
Creating a Project	1-2
Writing Code for Your Application	1-3
Connecting Objects with Interface Builder	1-3
Adding Other Resource Files	1-4
Choosing Document Extensions for Your Application	1-4
Compiling Your Program	1-4
Debugging Your Program	1-5
Adding Help to Your Application	1-5
Translate Your User Interface	1-5
Making Your Application Available to Users	1-6
2. Using Project Builder	2-1
Creating and Maintaining Projects in Project Builder	2-3

Creating a New Project	2-3
Opening an Existing Project	2-5
Opening and Converting Older Project Types	2-6
Creating a New Subproject	2-7
Setting Project Attributes	2-8
Application Attributes	2-9
Subproject Attributes	2-11
Bundle Attributes	2-11
Tool Attributes	2-12
Palette Attributes	2-13
Library Attributes	2-14
Managing Project Files	2-14
Adding Files to a Project	2-17
File Display Shortcuts	2-17
Building the Project	2-18
Build Targets	2-23
The Preamble and Postamble Files	2-24
Defining User-configurable Macros for a Project with IDL Interfaces	2-27
Setting Preferences	2-29
Build Defaults Controls	2-29
Tools Controls	2-29
Sounds Controls	2-31
Build Service Controls	2-31

Save Options Controls	2-31
Running and Debugging an Application	2-32
Running	2-32
Debugging	2-32
Project Builder Command Reference	2-33
Commands in the Project Menu	2-33
Commands in the Files Menu	2-34
3. Working with Interface Builder	3-1
An Orientation	3-3
Building an Application with Interface Builder	3-4
Specifying Object Attributes	3-4
Interconnecting Objects	3-5
Adding Code to Your Application	3-5
Composing the Interface	3-6
Opening a Nib File	3-6
When Interface Builder Starts	3-8
The Palette Window	3-9
The Interface Window	3-9
The Nib File Window	3-10
The Inspector Panel	3-21
Creating a Nib File	3-22
Saving the Nib File	3-23
What Is in a Nib File	3-24
Archived Objects	3-24

Sounds and Images	3-25
Class References	3-25
Connection Information	3-26
When You Load a Nib File	3-27
Using the Palettes	3-27
The Menus Palette	3-28
The Views Palette	3-28
The TextViews Palette	3-30
The Windows Palette	3-30
Adding an Object from a Palette to Your Interface	3-31
Placing Interface Objects.	3-33
Selecting Multiple Objects	3-34
Where Palette Objects Go	3-34
Sizing Windows and Panels	3-36
Initializing Text	3-38
Sizing Interface Objects.	3-39
Positioning and Sizing Precisely	3-40
Duplicating Objects.	3-42
Moving Objects to Other Windows	3-43
Copying Objects to Other Interfaces	3-44
Arranging Objects	3-45
Using the Alignment Panel	3-45
Making Columns and Rows of Objects	3-48
Removing Objects	3-49

The Coordinate System in Interface Builder.	3-49
OpenStep's Basic UI Design Philosophy.	3-51
Make It Consistent	3-51
Make it Feel Natural	3-51
Put the User in Charge.	3-51
Focus on the Mouse	3-52
Making Interface Objects the Same Size	3-52
Shrinking Objects to their Minimum Size.	3-54
Grouping Objects.	3-55
Layering Objects	3-57
Creating Matrices of Objects.	3-58
Creating Menus	3-60
Deleting a Menu Cell	3-61
Changing Titles of Menu Cells	3-61
Resequencing Menu Cells and Assigning Command Key Equivalents	3-61
Custom Menus	3-62
Setting Object Attributes.	3-63
Examining an Object's Attributes	3-65
Customizing Windows and Panels	3-67
Window Backing	3-68
Changing Class and Custom Windows.	3-69
Window Controls	3-69
Window Options	3-70

What is the Difference Between a Window and a Panel? . . .	3-70
Setting Button Attributes	3-71
The Anatomy of a Button	3-72
Titles and Icons	3-73
Key Equivalent	3-73
Button Type	3-74
Button Options	3-74
Associating Sounds and Images with Buttons	3-75
Managing Sounds and Images	3-78
Customizing Titles, Text Fields, and Scroll Views	3-81
Setting Textual Attributes	3-84
Setting Box (Group) Attributes	3-85
Customizing Browsers	3-87
Setting Attributes of Menu Cells and Pop-up Buttons	3-89
Pop-Up Lists and Pull-Down Lists	3-90
Compound Objects	3-91
NSControl and NSActionCell	3-91
Matrices	3-92
Special Compound Objects	3-93
Setting Matrix Attributes	3-95
Matrix Selection Mode	3-95
Cell Prototype	3-96
Automatically Resizing Objects	3-98
When There Are Conflicts	3-101

Some Effects of Automatic Resizing	3-102
Automatic Resizing: An Example	3-104
Using Tags	3-107
Making and Managing Connections	3-109
Communicating With Other Objects: Outlets and Actions.	3-109
Outlets	3-109
Delegates	3-110
Targets	3-111
Actions.	3-111
Connecting Objects	3-113
Outlet Connections	3-114
Action Connections	3-116
Connections Within the Interface.	3-118
Making Connections in Outline Mode	3-120
Examining Connections	3-122
Identifying Objects in Outline Mode	3-127
Enabling Inter-Field Tabbing	3-128
Disconnecting Objects.	3-130
Attaching Help to Objects.	3-132
Reviewing Help Attachments	3-135
Testing the Interface.	3-136
Creating a Class	3-137
Naming a New Class.	3-141
A Perspective on Class Hierarchy	3-142

Specifying Outlets and Actions	3-143
Adding Outlets	3-144
Adding Actions	3-145
Creating an Instance of Your Class	3-147
Connecting Your Class's Outlets	3-149
Connecting Your Class's Actions	3-151
Generating Source Code Files	3-153
Implementing a Subclass of <code>NSObject</code>	3-155
Making Your Class a Delegate	3-156
Implementing an <code>NSView</code> Subclass	3-158
Adding Existing Classes to Your Nib File	3-162
Updating a Class Definition	3-163
Adding IDL Template Objects to Your Interface	3-164
Connecting IDL Template Objects	3-165
Outlet Autotyping	3-166
Setting Preferences	3-167
General Preferences	3-167
Palettes Preferences	3-168
Adding Custom Palettes, Inspectors, and Editors	3-169
Interface Builder Command Reference	3-171
Commands in the Document Menu	3-171
Commands in the New Module Submenu	3-172
Commands in the Edit Menu	3-173
Commands in the Format Menu	3-173

Commands in the Group Submenu	3-175
Commands in the Align Submenu	3-175
Commands in the Size Submenu	3-177
Commands in the Tools Menu	3-177
Commands on the Palettes Submenu	3-178
4. Using Edit in Developer Mode	4-1
Starting Edit	4-1
Setting Preferences	4-3
User Options	4-4
Start-up Options	4-4
New Document Format Options	4-5
Default Font for RTF Files	4-5
Default Font for ASCII Files	4-5
Global Options	4-6
Save Options	4-6
Default Window Size Options	4-7
Emacs Key Bindings	4-7
Temporary Settings	4-7
Line Wrap Options	4-8
Rich Text Display Options	4-8
Text Options	4-8
Automatic Indenting Options	4-9
Structure Level of Blank Lines	4-9
Alignment Options	4-9

Open at Structure Level Options	4-10
Editing Modes	4-10
C Options	4-11
Structure for Top Level	4-11
Structure Level of Blank Lines	4-11
Tags Path	4-12
Include Path	4-12
Performing Basic Operations	4-13
Opening Edit Files	4-13
Using File Windows and Folder Windows	4-14
Contracting and Expanding Text in a File Window	4-15
Adding Help Links	4-17
Using Templates	4-18
Using Keyboard Editing Commands	4-21
Interacting with UNIX	4-21
Piping UNIX Output to a File	4-22
Using a Tags File	4-23
Edit Command Reference	4-24
Commands in the Main Menu	4-24
Commands in the File Menu	4-25
Commands in the Edit Menu	4-26
Commands in the Link Submenu	4-26
Commands in the Find Submenu	4-27
Commands in the Format Menu	4-28

Commands in the Font Submenu	4-28
Commands in the Text Submenu	4-29
Commands in the Help Submenu	4-30
Commands in the Structure Submenu	4-31
Commands in the Utilities Menu	4-31
Commands in the Expert Submenu	4-34
5. Using Icon Builder to Create Application Icons	5-1
Creating, Opening, and Saving Documents	5-1
Creating a New Document	5-2
Opening an Existing Document	5-3
Saving a Document	5-3
Editing Icon Documents	5-3
Using Icon Builder Tools	5-4
The Brush Tool	5-5
The Line Tool	5-5
The PaintBucket Tool	5-6
The Pencil Tool	5-6
The Rectangle Tool	5-6
The Selection Tool	5-7
The Text Tool	5-7
Using the Tools Inspector	5-8
The Brush Inspector	5-8
The Line Inspector	5-9
The Oval Inspector	5-9

The Pencil Inspector	5-10
The Rectangle Inspector	5-11
The Selection Inspector	5-12
The TextTool Inspector	5-14
Zooming In on a Document	5-15
Changing the Attributes of a Document	5-17
Working with Multiple-Icon Documents	5-17
Icon Builder Command Reference	5-18
Commands in the Main Menu	5-18
Commands in the Document Menu	5-18
Commands in the Format Menu	5-19
Commands in the Tools Menu	5-20
6. Navigating the OpenStep API with Header Viewer	6-1
Header Viewer and Header Files	6-2
Precompiled Headers	6-2
Language Elements	6-3
Header Viewer and OpenStep Documentation	6-4
Using Header Viewer	6-5
The Browser View	6-5
The Finder View	6-10
Adding Header Files	6-12
The Find Panel	6-13
Header Viewer and the File Viewer	6-13
Header Viewer and Edit	6-13

Setting Preferences.	6-14
Header Files Preferences.	6-14
Documentation Preferences	6-15
Other Options Preferences	6-15
Header Viewer Command Reference	6-16
Commands in the Find Menu.	6-16
Commands in the Utilities Menu.	6-18
7. The NSObject Class.	7-1
Class Description	7-1
Initializing an Object to Its Class	7-2
Instance and Class Methods.	7-2
Initializing the Class	7-3
Creating and Destroying Instances	7-4
Identifying Classes	7-5
Testing Class Functionality.	7-5
Testing Protocol Conformance	7-6
Obtaining Method Information	7-6
Describing Objects.	7-7
Posing	7-7
Error Handling.	7-7
Sending Deferred Messages	7-8
Forwarding Messages	7-8
Archiving	7-9

8. The Objective C Language	8-1
Objects	8-2
The id Data Type	8-3
Dynamic Typing	8-4
Messages	8-5
Polymorphism	8-6
Dynamic Binding	8-7
Classes	8-8
Inheritance	8-9
The NSObject Class	8-10
Inheriting Instance Variables	8-11
Inheriting Methods	8-11
Overriding One Method with Another	8-11
Abstract Classes	8-12
Class Types	8-13
Static Typing	8-13
Type Introspection	8-14
Class Objects	8-14
Creating Instances	8-16
Customization with Class Objects	8-16
Variables and Class Objects	8-18
Initializing a Class Object	8-19
Methods of the Root Class	8-19
Class Names in Source Code	8-20

Defining a Class	8-21
The Interface	8-21
Importing the Interface	8-23
Referring to Other Classes	8-24
The Role of the Interface	8-24
The Implementation	8-25
Referring to Instance Variables	8-26
The Scope of Instance Variables	8-28
How Messaging Works	8-31
Selectors	8-34
Methods and Selectors	8-35
Method Return and Argument Types	8-35
Varying the Message at Run Time	8-36
The Target-Action Paradigm	8-36
Avoiding Messaging Errors	8-37
Hidden Arguments	8-38
Messages to <code>self</code> and <code>super</code>	8-39
An Example	8-40
Using <code>super</code>	8-42
Redefining <code>self</code>	8-43
9. The Objective C Extensions	9-1
Categories	9-1
Adding to a Class	9-2
How Categories are Used	9-3

Categories of the Root Class	9-4
Protocols	9-5
How Protocols are Used	9-5
Methods for Others to Implement	9-6
Anonymous Objects	9-7
Nonhierarchical Similarities	9-9
Informal Protocols	9-9
Formal Protocols	9-10
Protocol Objects	9-12
Conforming to a Protocol	9-12
Type Checking	9-13
Protocols within Protocols	9-14
Remote Messaging	9-15
Distributed Objects	9-16
Language Support	9-18
Synchronous and Asynchronous Messages	9-18
Pointer Arguments	9-19
Proxies and Copies	9-21
Static Options	9-22
Static Typing	9-23
Type Checking	9-24
Return and Argument Types	9-25
Static Typing to an Inherited Class	9-25
Getting a Method Address	9-27

Getting an Object Data Structure	9-28
Type *Encoding	9-29
A. Debugging an OpenStep Application.	A-1
Debugger Objective C Support	A-1
Dynamic Types.	A-2
Finding Methods and Using Method Names in Non-expression Commands.	A-2
Setting Breakpoints	A-2
Calling Objective C Methods	A-3
Recovering from a Run-time System Crash.	A-3
Sample .dbxrc File.	A-3
Helpful User Default Variables to Set with <code>dwrite</code>	A-13
Tracing Objective C Objects	A-13
Invoking <code>messageSendDebug</code> Using <code>dwrite</code> Commands	A-14
Adding Individual Message Filters	A-14
Controlling Call Level Indentation	A-15
Invoking <code>messageSendDebug</code> from a Program or the Debugger	A-15
Enabling <code>messageSendDebug</code>	A-15
Adding Filters.	A-16
Controlling Call Level Indentation	A-17
Removing Filters	A-17
Disabling Filters	A-17
Setting a Breakpoint on a Filter Match	A-17

Examples	A-18
Example 1:	A-18
Example 2:	A-18
Example 3:	A-19
Implementing Your Own Filtering Mechanism	A-19
Debugging Applications Using Optimized Libraries	A-20
B. Interface Builder Application Programming Interface	B-1
Interface Builder's Design	B-2
The Object Hierarchy	B-3
Class References	B-3
Connection Information	B-4
Interface Builder's Programming Interface	B-5
Classes	B-5
Protocols	B-6
Other Programming Interfaces	B-6
C. Interface Builder API Classes	C-1
IBInspector	C-1
Class Description	C-1
Instance Variables	C-2
Method Types	C-3
Instance Methods	C-3
object	C-3
ok:	C-3
okButton:	C-4

revert:	C-4
revertButton:	C-5
textDidBeginEditing:	C-5
touch:.....	C-5
wantsButtons	C-5
window	C-5
IBPalette	C-6
Class Description.....	C-6
Instance Variables	C-7
Method Types.....	C-8
Instance Methods.....	C-8
associateObject:ofType:withView:	C-8
imageName:.....	C-8
finishInstantiate.....	C-9
originalWindow.....	C-9
paletteDocument.....	C-9
NSApplication Additions.....	C-9
Category Description	C-9
Instance Methods.....	C-10
connectDestination.....	C-10
connectSource	C-10
displayConnectionBetween:and:.....	C-10
isConnecting	C-10
stopConnecting.....	C-11

NSObject Additions.....	C-11
Category Description	C-11
Instance Methods	C-12
awakeFromDocument:.....	C-12
canSubstituteFor Class:.....	C-12
connectInspectorClassName	C-12
editorClassName.....	C-12
helpInspectorClassName.....	C-13
imageForViewer:.....	C-13
inspectorClassName	C-13
sizeInspectorClassName.....	C-13
NSCellAdditions	C-14
Category Description	C-14
Instance Methods	C-14
cellWillAltDragWithSize:.....	C-14
maximumSizeForCellSize:.....	C-14
minimumSizeForCellSize:.....	C-15
NSView Additions.....	C-15
Category Description	C-15
Instance Methods	C-16
allowsAltDragging:.....	C-16
maximumSizeFromKnobPosition:.....	C-16
minimumSizeFromKnobPosition:.....	C-16
placeView:.....	C-17

D. Interface Builder API Protocols	D-1
IB	D-1
Protocol Description	D-1
Method Types.....	D-2
Instance Methods.....	D-2
activeDocument	D-2
isTestingInterface.....	D-2
selectionOwner	D-2
IBConnectors	D-3
Protocol Description	D-3
Method Types.....	D-4
Instance Methods.....	D-4
destination.....	D-4
establishConnection	D-4
label.....	D-4
nibInstantiate	D-5
replaceObject:withObject:	D-6
source	D-6
IBDocuments	D-6
Protocol Description	D-6
Method Types.....	D-7
Instance Methods.....	D-8
addConnector:.....	D-8
attachObject:toParent:.....	D-8

attachObjects:toParent:.....	D-9
connectorsForDestination:	D-9
connectorsForDestination:ofClass:	D-9
connectorsForSource:	D-10
connectorsForSource:ofClass:	D-10
containsObject:	D-10
containsObjectWithName:forParent	D-11
copyObject:type:toPasteboard:.....	D-11
copyObjects:type:toPasteboard:.....	D-11
detachObject:.....	D-12
detachObjects:	D-12
documentPath	D-12
drawObject:.....	D-13
editor:didCloseForObject:	D-13
editorForObject:create:.....	D-13
nameForObject:	D-13
objects:	D-13
openEditorForObject:	D-14
parentOfObject:.....	D-14
pasteType:fromPasteboard:parent:	D-14
removeConnector:	D-14
replaceObject:withObject:	D-15
resignSelectionForEditor:	D-15
setName:forObject:.....	D-15

setSelectionFromEditor:.....	D-15
touch.....	D-16
IBEditors	D-16
Protocol Description	D-16
Method Types.....	D-17
Instance Methods.....	D-18
acceptsTypeFromArray:.....	D-18
activate	D-18
close.....	D-19
closeSubeditors	D-19
copySelection	D-19
deleteSelection.....	D-20
document	D-20
editedObject	D-20
initWithObject:inDocument:	D-20
makeSelectionVisible:.....	D-21
openSubeditorForObject:	D-21
orderFront.....	D-21
pasteInSelection.....	D-21
resetObject:	D-22
selectObjects:	D-22
validateEditing.....	D-22
wantsSelection	D-22
window	D-22

IBSelectionOwners.....	D-23
Protocol Description	D-23
Instance Methods.....	D-23
drawSelection	D-23
selection:.....	D-23
selectionCount	D-24
E. Interface Builder API Types and Constants	E-1
Symbolic Constants	E-1
Control Point Constants	E-1
Synopsis.....	E-1
Description	E-1
Global Variables	E-2
Notification Types	E-2
Synopsis.....	E-2
Description	E-2
Synopsis.....	E-3
Description	E-3
Pasteboard Types.....	E-3
Synopsis.....	E-3
Description	E-3

Figures

Figure 2-1	New Project Panel	2-4
Figure 2-2	Project Window	2-5
Figure 2-3	Open Panel	2-6
Figure 2-4	Project Conversion Attention Panel	2-7
Figure 2-5	New Subproject Panel.	2-8
Figure 2-6	Attributes Display	2-8
Figure 2-7	Project Name, Language, and Installation Directory	2-9
Figure 2-8	Information about the Main File	2-9
Figure 2-9	Application Icon Well	2-10
Figure 2-10	Document Icons and Extensions Well	2-10
Figure 2-11	System File Types List.	2-11
Figure 2-12	Subproject Attribute Controls	2-11
Figure 2-13	Standalone Bundle Attribute Controls.	2-12
Figure 2-14	Bundle Attribute Controls	2-12
Figure 2-15	Standalone Tool Attribute Controls	2-13
Figure 2-16	Tool Attribute Controls.	2-13

Figure 2-17	Palette Attribute Controls	2-13
Figure 2-18	Library Attribute Controls	2-14
Figure 2-19	Files Display in the Project Window.	2-15
Figure 2-20	Builder Display in Project Window	2-18
Figure 2-21	Target Pop-up List.	2-19
Figure 2-22	Options Button.	2-19
Figure 2-23	Build Options Panel	2-21
Figure 2-24	Build Button	2-22
Figure 2-25	Warnings and Error Messages.	2-22
Figure 2-26	Build Defaults Controls	2-29
Figure 2-27	Tools Controls	2-30
Figure 2-28	Sounds Controls.	2-31
Figure 2-29	Build Service Controls	2-31
Figure 2-30	Save Options Controls	2-31
Figure 2-31	Run Button	2-32
Figure 2-32	Debug Button.	2-32
Figure 3-1	Interface Builder and Your Application.	3-3
Figure 3-2	Opening a Nib File in the Project Builder Window	3-7
Figure 3-3	Opening a Nib File in the Open Panel	3-8
Figure 3-4	The Palette Window	3-9
Figure 3-5	An Interface Window	3-10
Figure 3-6	The Nib File Window	3-11
Figure 3-7	Icon Mode of Instances Display	3-12
Figure 3-8	Outline Mode of Instances Display.	3-13
Figure 3-9	Classes Display	3-14

Figure 3-10	Images Display	3-15
Figure 3-11	Sounds Display	3-17
Figure 3-12	IDL Display	3-18
Figure 3-13	File's Owner Icon	3-19
Figure 3-14	First Responder Icon	3-20
Figure 3-15	The Inspector Panel	3-21
Figure 3-16	New Info Panel	3-23
Figure 3-17	Saving a Nib File	3-24
Figure 3-18	Archived Objects	3-25
Figure 3-19	Sounds and Images	3-25
Figure 3-20	Custom Class Information	3-26
Figure 3-21	Connection Information	3-26
Figure 3-22	The Menus Palette	3-28
Figure 3-23	The Views Palette	3-29
Figure 3-24	The TextViews Palette	3-30
Figure 3-25	The Windows Palette	3-31
Figure 3-26	Dragging an Object from a Palette to the Application Interface	3-32
Figure 3-27	Placing an Interface Object	3-33
Figure 3-28	Putting a Panel in the Workspace	3-34
Figure 3-29	Putting NSViews and NSTextView in aWindow	3-35
Figure 3-30	Putting a Menu Cell in the Application's Menu	3-35
Figure 3-31	Sizing a Window with the Resize Bar	3-36
Figure 3-32	Sizing a Window with the Size Display of the Inspector Panel	3-37
Figure 3-33	Editing the Text on an NSButton (Switch) Object	3-38
Figure 3-34	Editing the Text on an NSMatrix Object	3-39

Figure 3-35	Sizing an Interface Object with its Resize Handles	3-40
Figure 3-36	Sizing an Interface Object with the Inspector Panel	3-41
Figure 3-37	Selecting an Object to Duplicate	3-42
Figure 3-38	The New Object After Duplication	3-43
Figure 3-39	Moving an Object to Another Window	3-44
Figure 3-40	Using the Alignment Panel	3-46
Figure 3-41	Using the Radio Buttons in the Alignment Panel	3-46
Figure 3-42	Aligning Objects Using the Grid	3-47
Figure 3-43	Aligning Objects to the Grid	3-47
Figure 3-44	Making a Column of Objects	3-48
Figure 3-45	Deleting a Object from the Interface	3-49
Figure 3-46	Interface Builder's Coordinate System	3-50
Figure 3-47	Selecting Several Objects and a Reference Object	3-53
Figure 3-48	The Objects Become the Same Size as the Reference Object . .	3-54
Figure 3-49	Sizing an <code>NSView</code> Object to Fit the Text It Contains	3-55
Figure 3-50	Selecting Objects and Using the Group Command	3-56
Figure 3-51	Using an <code>NSBox</code> Object to Group Objects	3-56
Figure 3-52	Layering Buttons in Front of an <code>NSScrollView</code> Object	3-58
Figure 3-53	Creating a Matrix of Radio Buttons	3-59
Figure 3-54	Adding a Menu Cell to the Application's Main Menu	3-60
Figure 3-55	Resequencing Menu Cells	3-61
Figure 3-56	Assigning a Command Key Equivalent	3-62
Figure 3-57	Using the Submenu Cell to Create a Custom Submenu	3-62
Figure 3-58	Attributes Display of <code>NSButton</code> Inspector	3-64
Figure 3-59	Selecting an Object in the Instances Display	3-66

Figure 3-60	Attributes Display for a Custom Class.	3-67
Figure 3-61	Attributes Display for Windows and Panels.	3-68
Figure 3-62	Window Controls	3-69
Figure 3-63	NSButton Attributes Display	3-72
Figure 3-64	Associating an Image with a Button.	3-75
Figure 3-65	Associating a Sound with a Button.	3-76
Figure 3-66	NSButton Attributes that Relate to Sounds or Images.	3-77
Figure 3-67	Adding a Sound or Image to a Nib File	3-79
Figure 3-68	Inspecting Sound Attributes	3-80
Figure 3-69	Inspecting Image Attributes.	3-81
Figure 3-70	Setting NSTextField Attributes	3-82
Figure 3-71	Setting NSScrollView Attributes.	3-83
Figure 3-72	Setting the Attributes of Text.	3-84
Figure 3-73	Font Panel	3-85
Figure 3-74	Setting NSBox Attributes	3-86
Figure 3-75	Setting NSBrowser Attributes	3-88
Figure 3-76	Setting NSPopUpButton Attributes.	3-90
Figure 3-77	Pop-up List's Trigger Button and Menu Cells	3-91
Figure 3-78	NSScrollView.	3-93
Figure 3-79	NSBrowser.	3-94
Figure 3-80	NSPopUpButton.	3-94
Figure 3-81	Setting NSMatrix Attributes.	3-95
Figure 3-82	Cell Prototype Inspector.	3-97
Figure 3-83	Size Inspector	3-99
Figure 3-84	Effects of Lines Inside and Outside the Autosizing Box	3-100

Figure 3-85	Specifying a Minimum Size for a Window	3-101
Figure 3-86	Resizing Example	3-102
Figure 3-87	Object A Resizes, Object B Does Not	3-103
Figure 3-88	Both Object A and Object B Resize	3-103
Figure 3-89	Original and Resized Windows.	3-104
Figure 3-90	Minimum Size Set for Window	3-105
Figure 3-91	Autosizing Behavior Set for Box	3-105
Figure 3-92	Autosizing Behavior Set for Button	3-106
Figure 3-93	Autosizing Behavior Set for Custom View	3-106
Figure 3-94	Specifying a Tag Integer for an Object	3-108
Figure 3-95	Outlet.	3-110
Figure 3-96	Action	3-112
Figure 3-97	Connecting Two Objects.	3-114
Figure 3-98	Inspecting an Outlet Connection.	3-115
Figure 3-99	Connecting Objects in the Instances Display.	3-116
Figure 3-100	Making an Action Connection.	3-117
Figure 3-101	Inspecting an Action Connection	3-118
Figure 3-102	Connecting Objects within an Interface	3-119
Figure 3-103	Connecting an Object in the Outline with an Object in the Interface	3-120
Figure 3-104	Displaying the Possible Connections	3-121
Figure 3-105	Making a Connection within the Nib FileWindow	3-122
Figure 3-106	Displaying the Outlets and Actions Associated with an Interface Object.	3-123
Figure 3-107	Examining a Connection through the Inspector Panel Connections Display	3-124

Figure 3-108	Checking Connections in the Instances Display	3-125
Figure 3-109	Looking at Connections Out in the Instances Display	3-126
Figure 3-110	Looking at Connections In in the Instances Display	3-126
Figure 3-111	Displaying an Image Representing the Object Selected in the Outline	3-127
Figure 3-112	Locating the Object in the Interface with an Arrow	3-128
Figure 3-113	Connecting Two NSForm Objects	3-129
Figure 3-114	Making the Connection in the Inspector Panel	3-130
Figure 3-115	Disconnecting Objects Using the Inspector Panel	3-131
Figure 3-116	Disconnecting Object in the Instances Display	3-132
Figure 3-117	Help Builder Panel	3-133
Figure 3-118	Help Display	3-135
Figure 3-119	Exiting Test Mode	3-137
	Class or <code>NSView</code>	
	Root Class or <code>NSView</code>	3-139
Figure 3-121	Flowchart for Integrating an Existing Class into an Application	3-140
Figure 3-122	Selecting and Subclassing a Superclass	3-141
Figure 3-123	Naming the New Class	3-142
Figure 3-124	Classes Display	3-143
Figure 3-125	Accessing the Outlets of a Class	3-144
Figure 3-126	Naming a New Outlet	3-145
Figure 3-127	Accessing the Actions of a Class	3-146
Figure 3-128	Naming a New Action	3-146
Figure 3-129	Instantiating a Custom Class	3-148
Figure 3-130	The New Instance in the Instances Display	3-149

Figure 3-131	Connecting an Outlet	3-150
Figure 3-132	Specifying the Outlet Identifier	3-151
Figure 3-133	Connecting an <code>NSControl</code> Object	3-152
Figure 3-134	Selecting the Action Method	3-153
Figure 3-135	Unparsing the Nib File	3-154
Figure 3-136	Unparse Attention Panel	3-155
Figure 3-137	Making Your Class a Delegate	3-157
Figure 3-138	An <code>NSView</code> Custom Class	3-159
Figure 3-139	Making an Instance of an <code>NSView</code> Subclass	3-160
Figure 3-140	Assigning a Class Name to your <code>NSView</code> Object	3-161
Figure 3-141	Dragging a Header File into Your Nib File	3-162
Figure 3-142	Updating the Nib File	3-163
Figure 3-143	Selecting the Class Definition to Update	3-164
Figure 3-144	Parse IDL Button (IDL Display)	3-165
Figure 3-145	Make Template Object Button (IDL Display)	3-165
Figure 3-146	Interface Builder's General Preferences Panel	3-168
Figure 3-147	Interface Builder's Palettes Preferences Panel	3-169
Figure 4-1	Edit Preferences Panel	4-3
Figure 4-2	Options Pop-up List	4-4
Figure 4-3	Edit Start-up Options	4-4
Figure 4-4	New Document Format Options	4-5
Figure 4-5	RTF Default Font	4-5
Figure 4-6	ASCII Default Font	4-6
Figure 4-7	Save Options	4-6
Figure 4-8	Default Window Size Options	4-7

Figure 4-9	Emacs Key Bindings Options.	4-7
Figure 4-10	Line Wrap Options	4-8
Figure 4-11	Rich Text Display Options	4-8
Figure 4-12	Automatic Indenting Options	4-9
Figure 4-13	Structure Level of Blank Lines in Text Options	4-9
Figure 4-14	Alignment Options	4-10
Figure 4-15	Open at Structure Level Options.	4-10
Figure 4-16	Editing Modes File Extensions	4-10
Figure 4-17	Structure for Top Level Options	4-11
Figure 4-18	Structure of Blank Lines in C Code Options	4-11
Figure 4-19	Tags Path.	4-12
Figure 4-20	Include Path	4-13
Figure 4-21	File Window with Only First-Level Text Expanded	4-15
Figure 4-22	File Window with Some Second-Level Text Expanded.	4-16
Figure 4-23	File Window with Some Third-Level Text Expanded	4-17
Figure 4-24	Expansion Dictionary Panel.	4-19
Figure 4-25	Add Button.	4-20
Figure 4-26	Remove Button.	4-20
Figure 5-1	New Document Panel	5-2
Figure 5-2	Tools Panel	5-4
Figure 5-3	The Brush Tool.	5-5
Figure 5-4	The Line Tool	5-5
Figure 5-5	The Oval Tool.	5-5
Figure 5-6	The PaintBucket Tool	5-6
Figure 5-7	The Pencil Tool.	5-6

Figure 5-8	The Rectangle Tool	5-6
Figure 5-9	The Selection Tool	5-7
Figure 5-10	The Text Tool	5-7
Figure 5-11	The Brush Inspector	5-8
Figure 5-12	The Line Inspector	5-9
Figure 5-13	The Oval Inspector	5-10
Figure 5-14	The Pencil Inspector	5-11
Figure 5-15	The Rectangle Inspector	5-12
Figure 5-16	The Selection Inspector	5-13
Figure 5-17	Flip Filter Attributes	5-13
Figure 5-18	Rotate Filter Attributes	5-14
Figure 5-19	Revert and Apply Buttons	5-14
Figure 5-20	The TextTool Inspector	5-15
Figure 5-21	The ObeseBits Panel	5-16
Figure 6-1	Header Viewer's Browser View	6-6
Figure 6-2	Removing a Header File	6-7
Figure 6-3	Selecting Direct Headers or All Headers in a Header Hierarchy	6-8
Figure 6-4	Choosing Display in a Class Hierarchy	6-9
Figure 6-5	Header Viewer's Finder View	6-10
Figure 6-6	Find Results List	6-11
Figure 6-7	Selecting Find Control Options	6-12
Figure 6-8	Header Viewer's Find in Viewer Panel	6-13
Figure 6-9	Header Viewer Preferences Panel	6-14
Figure 6-10	Documentation Directories Panel	6-15
Figure 6-11	Other Options Panel	6-16

Figure 8-1	Some NSMatrix Objects	8-2
Figure 8-2	Some Application Kit Classes	8-9
Figure 8-3	Inheritance Hierarchy for Cells	8-17
Figure 8-4	The Scope of Instance Variables	8-29
Figure 8-5	Messaging Framework	8-33
Figure 8-6	High, Mid, and Low	8-41
Figure 9-1	Remote Messages.	9-17
Figure 9-2	Round-Trip Message.	9-19

Tables

Table 2-1	Standard Types of Projects.....	2-2
Table 2-2	Project Builder Modes.....	2-5
Table 2-3	Categories of Project Files	2-16
Table 2-4	Build Options.....	2-20
Table 2-5	Build Targets	2-23
Table 2-6	Additional Build Targets	2-26
Table 2-7	Project Menu Commands.....	2-33
Table 2-8	File Menu Commands.....	2-34
Table 3-1	Object Attributes and Messages	3-65
Table 3-2	Window Options	3-70
Table 3-3	Button Types	3-74
Table 3-4	Button Options.....	3-74
Table 3-5	Title, Text Field, and Scroller Options	3-83
Table 3-6	Browser Options	3-88
Table 3-7	Cells Options	3-98
Table 3-8	Flow Chart Legend	3-138

Table 3-9	Document Menu Commands	3-171
Table 3-10	New-Module Menu Commands	3-172
Table 3-11	Edit Menu Commands	3-173
Table 3-12	Format Menu Commands	3-174
Table 3-13	Group Submenu Commands	3-175
Table 3-14	Align Submenu Commands	3-176
Table 3-15	Size Submenu Commands	3-177
Table 3-16	Tools Menu Commands	3-178
Table 3-17	Palettes Menu Commands	3-178
Table 4-1	Edit Command-line Options	4-2
Table 4-2	Keyboard Editing Commands	4-21
Table 4-3	Arguments for UNIX Commands	4-23
Table 4-4	File Menu Commands	4-25
Table 4-5	Edit Menu Commands	4-26
Table 4-6	Link Submenu Commands	4-26
Table 4-7	Find Submenu Commands	4-27
Table 4-8	Format Menu Commands	4-28
Table 4-9	Text Submenu Commands	4-29
Table 4-10	Help Submenu Commands	4-30
Table 4-11	Structure Submenu Commands	4-31
Table 4-12	Utilities Menu Commands	4-32
Table 4-13	Expert Menu Commands	4-34
Table 5-1	Icon Builder Submenus	5-18
Table 5-2	Icon Builder's Document Submenu	5-19
Table 5-3	Icon Builder's Format Menu	5-19

Table 5-4	Icon Builder's Tools Menu	5-20
Table 6-1	Language Elements You Can Look at with Header Viewer . .	6-3
Table 6-2	Find Menu Commands	6-17
Table 6-3	Utilities Menu Commands	6-18
Table 8-1	Scope Levels for Instance Variables	8-29
Table 9-1	Type Codes	9-29
Table 9-2	Additional Encodings	9-31

Preface

This manual, *OpenStep Development Tools*, describes the essential tools for developing an application using WorkShop™ OpenStep™ —the Project Builder, Interface Builder, Header Viewer, Icon Builder, and Edit applications. The manual also includes chapters on the Objective C language and the NSObject class.

Who Should Use This Book

If you are developing or designing OpenStep applications, this book will help you understand how to use the WorkShop OpenStep development tools to create a project and an application interface, and build and debug the project.

Before You Read This Book

This manual assumes you are familiar with the standard Solaris™ OpenStep™ user interface. Before attempting to develop an OpenStep application, you should be familiar with the workspace, the dock, the File Viewer, the applications supplied with Solaris OpenStep and the general look and feel of the OpenStep desktop. If you have not used Solaris OpenStep, the following end-user manuals will help you learn about it:

- *Quick Start to Using the OpenStep Desktop*
- *Using the OpenStep Desktop*

If you are developing an OpenStep application with a graphical interface you also need to be familiar with the guidelines covered in *OpenStep User Interface Guidelines*.

How This Book Is Organized

This manual contains the following chapters and appendices:

Chapter 1, “Introduction,” provides an overview of the tools and techniques that you’ll use to assemble a working application. The tools introduced in this chapter are discussed in greater detail in other chapters of this manual

Chapter 2, “Using Project Builder,” describes the central control point for application development in WorkShop OpenStep. Project Builder helps you with each stage of application development, from inception to installation.

Chapter 3, “Working with Interface Builder,” describes the tool that lets you assemble your application’s user interface (and other parts) from predefined building blocks, and lets you create new building blocks of your own design.

Chapter 4, “Using Edit in Developer Mode,” describes the OpenStep text editor you will be using to edit and debug your application’s source files. This chapter emphasizes the developer mode options of Edit. The user mode of Edit and the features available in both modes are described in *Using the OpenStep Desktop*.

Chapter 5, “Using Icon Builder to Create Application Icons,” describes a simple graphic editor for creating and editing application icons.

Chapter 6, “Navigating the OpenStep API with Header Viewer,” describes an OpenStep application that you can use to browse classes, language elements, and header files, and to perform searches on header files.

Chapter 7, “The NSObject Class,” briefly describes NSObject, the root class of all ordinary Objective C inheritance hierarchies.

Chapter 8, “The Objective C Language,” describes the OpenStep Objective C language as well as the principles of object-oriented programming as implemented in Objective C.

Chapter 9, “The Objective C Extensions,” describes more advanced and less commonly used features of Objective C.

Appendix A, “Debugging an OpenStep Application,” provides information on using the SPARCworks Debugger and other tools to debug an OpenStep application.

Appendix B, “Interface Builder Application Programming Interface,” describes the application programming interface (API) that lets you build custom palettes, inspectors, and editors for Interface Builder.

Appendix C, “Interface Builder API Classes,” describes the classes in the Interface Builder API.

Appendix D, “Interface Builder API Protocols,” describes the protocols in the Interface Builder API.

Appendix E, “Interface Builder API Types and Constants,” describes the types and constants in the Interface Builder API.

Related Books

For information on OpenStep classes, refer to *OpenStep Programming Reference*.

What Typographic Changes Mean

Table P-1 describes the typographic changes used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% You have mail.</code>
AaBbCc123	What you type, contrasted with on-screen computer output	<code>machine_name% su</code> Password:
<i>AaBbCc123</i>	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
<i>AaBbCc123</i>	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be root to do this.

Shell Prompts in Command Examples

Table P-2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-2 Shell Prompts

Shell	Prompt
C shell prompt	<code>machine_name%</code>
C shell superuser prompt	<code>machine_name#</code>
Bourne shell and Korn shell prompt	<code>\$</code>
Bourne shell and Korn shell superuser prompt	<code>#</code>

Introduction



There are a number of ways you might draw a distinction between programs and applications. Programs are simple; applications are complicated. Programs are small; applications are large. Programs run from a command line; applications have graphical user interfaces. A program has just a few source files; an application may have many.

No matter how you draw the line, as you move from writing programs to developing applications, you need to focus increasing attention on project management. If the application is the end result, the project is how you get there. The project can be thought of as both the steps you go through and the source files you use to construct an application.

A complete project management strategy includes strategies for creating, organizing, and maintaining source files; building the application from its sources; running and debugging the application; revising the source files to fix bugs; and installing the finished application—or preparing it for others to install.

In the WorkShop™ OpenStep™ development environment, the hub of application development is Project Builder—a project manager that is itself an OpenStep application. Project Builder is not the only tool you use to manage your project and develop your application. Instead, it is like the control center from which you switch from one application development task to another, and from one tool to another.

Putting Together an OpenStep Application

This introduction takes a brief look at the components of an OpenStep application. It explains the path that Project Builder and other WorkShop OpenStep tools offer you for going from a set of source files to a working application. It looks at the application development process in terms of resources and tasks that you, the developer, must provide and those that Project Builder and other OpenStep tools provide for you. Subsequent chapters present detailed information for each of the tools introduced here.

The process of developing an application can be divided into three general tasks: designing, coding, and debugging. These tasks are never performed entirely sequentially. You may decide after some coding that you need to change some aspect of design. Debugging always reveals code that needs rewriting, and occasionally exposes design flaws. When you develop an application with OpenStep tools, you can move easily among these tasks.

The following sections enumerate the components of the OpenStep application development process, describing those portions for which you are responsible and those which Project Builder, Interface Builder, and other WorkShop OpenStep development tools handle for you. For more information on Project Builder, see Chapter 2, “Using Project Builder”; for more on Interface Builder, see Chapter 3, “Working with Interface Builder.”

Designing Your Application

Before you write any code, you should spend some time thinking about design. Some components of application design to consider are functionality, program structure, and user interface. You should think about the goals of your application and the techniques you might use to meet those goals. You should determine the unique classes that your application requires and think about how to divide your program into separate modules. You should sketch out user interface ideas, and use Interface Builder to prototype and test those ideas.

Creating a Project

With the basic design determined, you can use Project Builder to start a new project.

In WorkShop OpenStep, a project is physically represented by a directory under the control of Project Builder; all of the components of the project must reside in this directory. When you start a new project, Project Builder automatically generates the project directory and a set of source files common to all applications, including a main file, a nib file, a makefile, and others.

The main file includes the standard `main()` function required in all C programs. The nib file is used by Interface Builder to archive the application's user interface. The makefile is updated by Project Builder to keep track of all the source files from which your application is built. Another file in the project directory, `PB.project`, is used by Project Builder itself to keep track of various project components.

Throughout the life of the project, you will add to and update the files in the project directory. WorkShop OpenStep development tools, including Project Builder and Interface Builder, may add to and maintain other files in this directory as your project grows.

Writing Code for Your Application

To establish the unique workings of your application, you create class interface and implementation files that include code for the appropriate methods and instance variables. Interface Builder can help in this process by creating skeletal code for a class if you list the methods in the Inspector panel. If you create the source files first, Interface Builder can parse them to learn about their `id` instance variables and action methods.

Project Builder lets you add source files to your project at any time. You can create other source files using standard C, Objective C, and C++ code. Project Builder can also know about and manage other files, such as `pswrap` files containing PostScript™ code within C function wrappers.

Connecting Objects with Interface Builder

In Interface Builder, you can interconnect objects in your application. For example, you can establish the target and action for a control in the interface.

Interface Builder puts information about the classes used by your application in the nib file; included are Application Kit classes and other classes provided by OpenStep, as well as the custom classes you define. The nib file contains all the information required to generate the objects in your application at run

time: specifications for objects, connections between objects, icons, sounds, and other features. An OpenStep application can have one or more nib files for each application you create.

Adding Other Resource Files

Resource files are frequently used to customize the user interface for your application. Project Builder allows you to add icons for both your application and its documents. Interface Builder allows you to add icons and sounds for the buttons in your user interface. You can put other images in your application using Application Kit classes and PostScript code. You can add other sounds using Sound Kit methods. Project Builder provides a drag-and-drop interface for adding sounds, images, and other resource files to your project, including unique icons for your application and its document files.

Choosing Document Extensions for Your Application

If your application reads and writes documents, you will need to take measures to see to it that the Workspace Manager knows about and can work with those files. First, you need to write file management code that saves the documents with a unique extension. You also need to use the Project Builder application's Attributes display to specify document extensions for an application. Project Builder adds these extensions to the appropriate file to assure that your application is invoked by Workspace Manager when the user double-clicks on a file with the specified extensions.

Compiling Your Program

As you add source files to your application, Project Builder lists them in the project `Makefile`. When you use its `Build` command, Project Builder starts the `make` utility, which in turn reads the project `Makefile` and generates the executable file from the sources. As the `make` utility runs, it issues system commands to compile and link your application's source files into an executable file. The project `Makefile`, generated by Project Builder, provides the information the `make` utility needs to do this job. The warnings generated by the compiler and link editor provide information to help you locate and fix bugs detected at compile time.

In building your project, the `make` utility keeps track of source updates. Each time you run the `make` utility, only the source files that have been updated since the last `make` are regenerated; the rest are used as is. This minimizes the time required to generate your executable file.

Once you start building your application, Project Builder provides an interactive interface to Edit for locating source code problems detected by the compiler and link editor. Anytime the compiler encounters an error, Edit can locate the code with a single click—you can then edit out the problem and begin compiling again.

Debugging Your Program

After you successfully compile your program, you are ready to try running it. The easiest way to do so is by choosing Debug in the Project Builder application's Builder display. This selection builds your application (if necessary), then starts the distributed Debugger. The Debugger is described in detail in the SPARCWorks manual *Debugging a Program*.

Adding Help to Your Application

Using Project Builder, Interface Builder, and Edit, you can create context-sensitive help for your application. The standard help template provided by Interface Builder includes general information on the OpenStep environment. You can add to this template to include application-specific help, and you can create links between the controls in your application and the help system to provide the user with context-specific assistance.

Translate Your User Interface

When the application is complete and help is available, you can create alternate versions with translated text for windows, panels, menu items, and buttons, as well as any help information you have added. The OpenStep application programming interface (API) provides ways of accessing bundles in your application containing the text and user interface in various languages you want to support. Chapter 2, "Using Project Builder," provides information on how to make a project localizable.

Making Your Application Available to Users

Once an application is debugged, you can install it in an application directory using Project Builder. Project Builder lets you determine which directory to install the application in and provides a way to automatically install the application when you build it.

When the user double-clicks on a document file, the Workspace Manager has to locate and start the executable file for that application. Workspace Manager looks for the executable file in a systematic sequence of directory paths. This search sequence is contained in an environmental variable `path`. You can place an application in any of the directories specified in `path`.

Because of the search sequence specified by `path`, you can replace an application located later in the sequence with one of the same name earlier in the sequence. For example, `$(HOME)/Apps` occurs before `/usr/openstep/Apps` in `path`; if you place an application in the directory `$(HOME)/Apps` with the same name as an application in the `/usr/openstep/Apps` directory, the Workspace Manager finds and starts the version in `$(HOME)/Apps` (the `Apps` subdirectory in your home directory). You should consider the path when naming and installing applications.



Project Builder is the hub of application development in the WorkShop™ OpenStep™ development environment. It manages the components of your application and gives you access to the other development tools you use to create and modify these components. Project Builder is involved in all stages of the development process, from providing you with the basic building blocks for a new application to installing the application when it is finished.

Project Builder's unit of organization is the project. A project can be defined in two ways: conceptually and physically. Conceptually, a project comprises a number of source components and is intended to produce a given end product, such as an application. (Other types of end products are possible, as described in Table 2-1 on page 2-2.) Physically, a project is a directory containing source files and Project Builder's controlling file, `PB.project`. This file records the components of the project, the intended end product, and other information. For a file to be part of a project, it must reside in the project directory and be recorded in the project's `PB.project` file. You do not edit `PB.project` directly; your actions in the Project Builder application—adding source files, modifying the project name or installation directory, and so on—have the effect of updating this file.

Project Builder can be used to create and maintain the standard types of OpenStep projects described in Table 2-1.

Table 2-1 Standard Types of Projects

Type of Project	Description
application	A standalone OpenStep application, such as the applications found in <code>/usr/openstep/Apps</code> .
subproject	A project within a project. With larger applications, it is often convenient to group components into subprojects, which can be built independently from the main project. In building a project, Project Builder builds the subprojects as needed and then uses their end products—usually <code>.o</code> files—to build the main project.
bundle	A directory containing resources that can be used by one or more applications. These resources might include such things as images, sounds, character strings, nib files, or executable code. For more information, see “NSBundle” in <i>OpenStep Programming Reference</i> . A bundle can be a standalone project, or contained within another project.
palette	A loadable palette that can be added to Interface Builder’s Palettes window.
tool	A tool is a command-line utility that has no resources, but that can be run by a user or by an application. An example is a server on another system with which an application might want to interact.
library	A directory containing executable code that can be used by one or more applications. A library is a standalone project, and is linked with other applications that use its executable code.

Project Builder also helps you prepare your application (or other type of project) for various language markets, a process called “localization.” It does this by helping you group language-dependent components of your application—TIFF and nib files, for example—in subdirectories of the project. These subdirectories are named for a language and have a `.lproj` extension (for example, `Spanish.lproj`), and so are commonly called `.lproj`

directories. Through the facilities of the `NSBundle` class, your application can load the appropriate, language-dependent components depending on the user's preferred language. (See "NSBundle" in *OpenStep Programming Interface*.)

You can start Project Builder (located in `/usr/openstep/Developer/Apps`) from the workspace as you would any other application, by double-clicking its icon in the workspace. When it starts, only the main menu is visible. Once Project Builder is running, you can create a new project or open an existing project as described below.

Creating and Maintaining Projects in Project Builder

This section describes how to create a new project in Project Builder and how to convert a NEXTSTEP project to the Sun™ project format. You will also find information here about maintaining your project.

Creating a New Project

To create a new project, choose the New command in the Project menu. A panel is displayed (see Figure 2-1 on page 2-4) in which you specify a path name and name for the project. Specify a new directory on the Name line, or choose an existing directory in the browser (and leave the name `PB.project` in the Name field) if you want to use that directory as the root of the new project.



Figure 2-1 New Project Panel

By default, the new project is a standalone application. A pop-up list in the panel lets you create a bundle, palette, tool, or library instead. No matter what type of project you create, a project window for the new project is displayed (see Figure 2-2 on page 2-5).

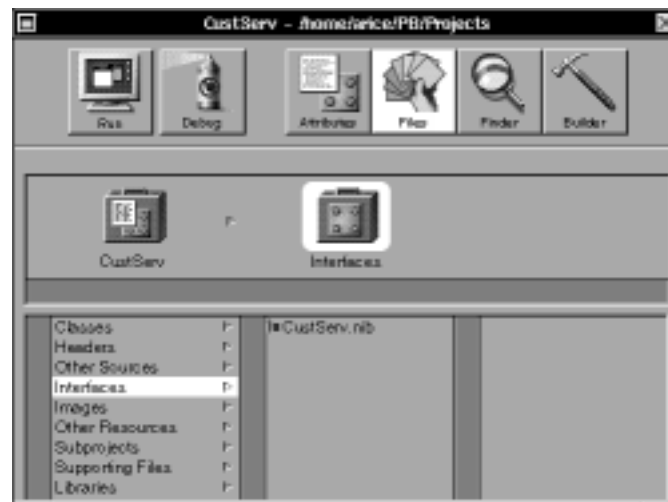


Figure 2-2 Project Window

You will use this project window to maintain, build, and debug the project, as described in the rest of this chapter. Table 2-2 lists the four modes of operation indicated by the four buttons in the upper right portion of the panel.

Table 2-2 Project Builder Modes

Mode	Purpose
Attributes	Set attributes of your project
Files	Add, remove, or open project files
Finder	Search for text in project files
Builder	Build the project

Opening an Existing Project

To open an existing project, choose the Open command in the Project menu. A standard Open panel is displayed in which you specify the project to open (see Figure 2-3 on page 2-6). Select the file named `PB.project` in the project directory and click on Open to open the project.

When you open a project, its project window is displayed in Project Manager.



Figure 2-3 Open Panel

Opening and Converting Older Project Types

To open an existing NEXTSTEP project that has not been converted to the Sun project format, choose the Open command in the Project menu. A standard Open panel is displayed in which you specify the project to open. Select the file named `PB.project` in the project directory and click on Open to open the project.

A panel is displayed warning you that the project file is a NEXTSTEP style project file that needs to be converted to a Sun style project file (see Figure 2-4 on page 2-7). Since the conversion process overwrites several project files, you are asked if you want to back up those files first before converting the project. Unless you are sure you do not need to do this, you should click on Backup First (or Cancel if you decide not to continue)—this causes the `PB.project` file and its associated makefiles to be saved in a directory named `savedNextFiles`.



Figure 2-4 Project Conversion Attention Panel

Once the project is converted, its project window is displayed in Project Manager. When you save the resulting project, it will be saved as a `PB.project` file in the same directory. This is the file you will open in the future when you work with the project. You might want to build your newly converted project with the `clean` target, to make sure that it gets rebuilt from scratch under Sun OpenStep.

The conversion process will convert the existing `PB.project` and associated makefiles to the OpenStep format. If you have already modified `Makefile.preamble` or `Makefile.postamble`, you will have to insert those modifications into the new versions of these files by hand.

Creating a New Subproject

To create a new subproject, open the project in which you want to create it and choose the `NewSubproject` command in the Project menu. A panel is displayed in which you specify a name and type for the new subproject (see Figure 2-5 on page 2-8). Specify a name for the subproject.

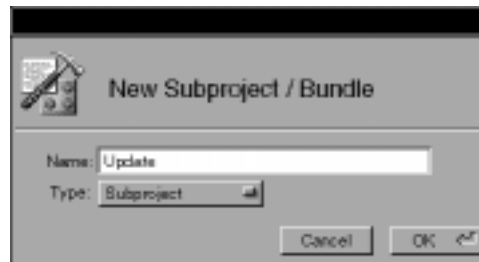


Figure 2-5 New Subproject Panel

A pop-up list in the panel lets you create a bundle or tool that is part of your project instead. No matter what type of subproject you create, the new subproject is displayed in the Subproject category in the project window.

Setting Project Attributes

To bring up the Attributes display shown in Figure 2-6, click on the Attributes button in the project window.



Figure 2-6 Attributes Display

The contents of the Attributes display vary depending on the type of project—application, bundle, palette, tool, or library. The contents of these five types of Attributes display are shown in the subsections that follow.

Application Attributes

If the project is an application, the Attributes display contains the controls for defining application attributes shown in Figure 2-7.




Project Type:	Application
Project Name:	CustServ
Language:	English
Install In:	\$(HOME)/openstep/Apps

Figure 2-7 Project Name, Language, and Installation Directory

This group of controls includes fields for specifying the project name, the primary language (that is, the language in which the project is being developed), and the target directory.

The Main File Info, shown in Figure 2-8, has fields for specifying the application class and the application's main nib file, plus an option for regenerating the Main file whenever you save the project. (Project Builder maintains this file and you are not expected to change it; therefore you should leave this option checked, unless there is a reason why you need to maintain the Main file yourself.)



Main File Info	
Generate Main File on Save	<input checked="" type="checkbox"/>
App. Class:	NSApplication
App. nib File:	CustServ

Figure 2-8 Information about the Main File

The Application Icon well, shown in Figure 2-9, displays the application icon. The default application (shown in Figure 2-9) is used if you do not provide one of your own choosing. To associate a new icon with the application, drag its TIFF file from the workspace into the well. The file is copied to the project directory, although it is not displayed in any of the categories shown in the Files display.



Figure 2-9 Application Icon Well

The Document Icons and Extensions well, shown in Figure 2-10, is where you indicate what types of documents your application is able to deal with. If you are creating your own document type, create a document icon for it and drag the TIFF file containing that icon into the well. Once the icon is in the well, change its label to match the document extension.

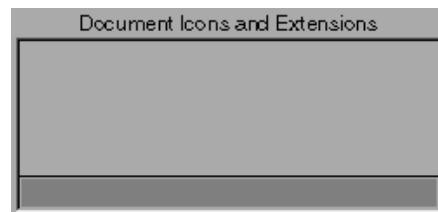


Figure 2-10 Document Icons and Extensions Well

The System File Types list, shown in Figure 2-11 on page 2-11, lists OpenStep file types (as identified by their standard OpenStep file extensions), any of which you may choose to have your application handle by selecting the file type in the scrolling list. When you select a file type by clicking on it, a check mark is displayed next to its name, and it is added to the Document Icons and Extensions well. Click on the file type again if you want to deselect it and remove it from the well.

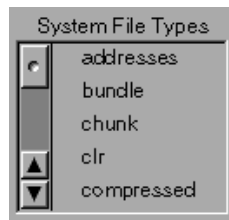


Figure 2-11 System File Types List

Subproject Attributes

If the project is a subproject, the Attributes display contains the controls shown in Figure 2-12 for defining project attributes.

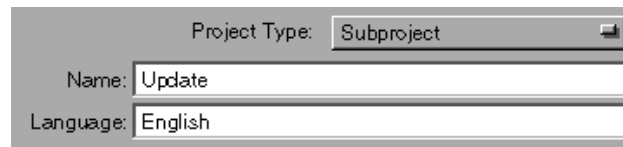


Figure 2-12 Subproject Attribute Controls

The Project Type pop-up list contains items that let you convert the subproject to a bundle or a tool.

The text fields allow you to change the project name and primary language.

Bundle Attributes

If the project is a standalone bundle, the Attributes display contains the controls shown in Figure 2-13 on page 2-12 for defining project attributes.



Figure 2-13 Standalone Bundle Attribute Controls

The text fields allow you to change the name, primary language, target directory, and extension.

If the project is a bundle that is part of another project, the Attributes display contains the controls shown in Figure 2-14 for defining project attributes.

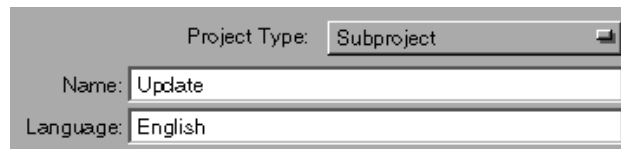


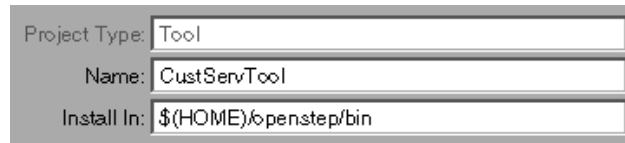
Figure 2-14 Bundle Attribute Controls

The Project Type pop-up list contains items that let you convert the bundle to a subproject or a tool. This is possible only with a bundle that is part of another project, not with a standalone bundle.

The text fields allow you to change the project name, primary language, and extension.

Tool Attributes

If the project is a standalone tool, the Attributes display contains the controls shown in Figure 2-15 on page 2-13 for defining project attributes.

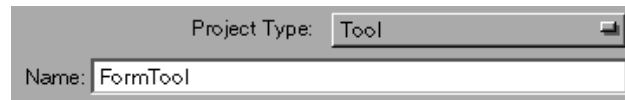


A screenshot of a dialog box titled "Standalone Tool Attribute Controls". It contains three text input fields. The first field is labeled "Project Type:" and contains the text "Tool". The second field is labeled "Name:" and contains the text "CustServTool". The third field is labeled "Install In:" and contains the text "\$\$(HOME)/openstep/bin".

Figure 2-15 Standalone Tool Attribute Controls

The text fields allow you to change the name and target directory.

If the project is a tool that is part of another project, the Attributes display contains the controls shown in Figure 2-16 for defining project attributes.



A screenshot of a dialog box titled "Tool Attribute Controls". It contains two controls. The first is a pop-up list labeled "Project Type:" with "Tool" selected. The second is a text input field labeled "Name:" containing the text "FormTool".

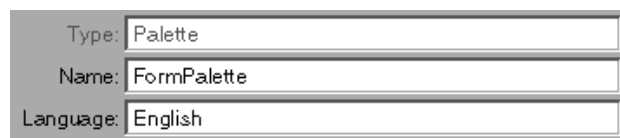
Figure 2-16 Tool Attribute Controls

The Project Type pop-up list contains items that let you convert the tool to a bundle or a subproject. This is possible only with a tool that is part of another project, not with a standalone tool.

The text field allows you to change the project name.

Palette Attributes

If the project is a palette, the Attributes display contains the controls shown in Figure 2-17 for defining project attributes.



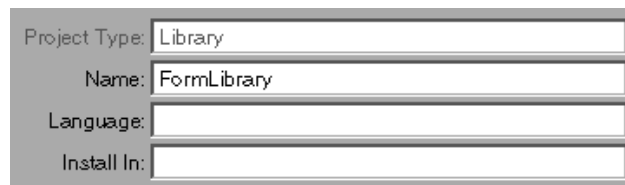
A screenshot of a dialog box titled "Palette Attribute Controls". It contains three text input fields. The first field is labeled "Type:" and contains the text "Palette". The second field is labeled "Name:" and contains the text "FormPalette". The third field is labeled "Language:" and contains the text "English".

Figure 2-17 Palette Attribute Controls

The text fields allow you to change the project name and the primary language.

Library Attributes

If the project is a library, the Attributes display contains the controls shown in Figure 2-18 for defining project attributes.



Project Type:	Library
Name:	FormLibrary
Language:	
Install In:	

Figure 2-18 Library Attribute Controls

The text fields allow you to change the project name, primary language, and target directory.

By default, Project Builder builds Solaris shared libraries for library projects. If you want to build a traditional `.a` style library archive, you must create your own `Makefile.preamble` containing the following macro override:

```
LIBRARY_STYLE = STATIC
```

See “Creating a `Makefile.preamble`” on page 2-27 for information on creating your own `Makefile.preamble`.

Managing Project Files

The Files display of the project window is used to manage the files in the project. You can use this display to add or delete project files, as well as open them for viewing or editing.

To bring up the Files display, shown in Figure 2-19 on page 2-15, click on the Files button in the project window.



Figure 2-19 Files Display in the Project Window

The Files display provides a file viewer similar to the Workspace Manager’s File Viewer, with categories of project components displayed in the left-hand column and project files for each category displayed to the right. These project categories do not correspond to project subdirectories—the categories are logical rather than physical groupings of files.

The project directory provides you and Project Builder with a convenient way to organize the files used in putting together your application. As shown here, files in the project directory are grouped by Project Builder into a number of categories. These categories are represented with a suitcase icon (and are frequently referred to as *suitcases*). These categories are described briefly in Table 2-3 on page 2-16.

Table 2-3 Categories of Project Files

Category	Description
Classes	Files containing code for custom classes used by an application.
Headers	Files containing declarations of methods and functions used by an application.
Other Sources	Files containing code (other than class code) for an application. These may include .m files (containing Objective C code), .c files (containing standard C code), .C or .cc files (containing C++ code), .psw files (containing PostScript™ code), and other sources. Project Builder automatically adds the file <i>ApplicationName_main.m</i> to Other Sources.
Interfaces	Nib files for each application and for each new module added to an application. The flag icon next to a file name in the Interfaces suitcase indicates that the file is localizable (that is, the file is in the <i>Language.lproj</i> subdirectory in the project directory, rather than in the project directory itself).
Images	Files containing images (other than icons) used by an application, including TIFF or EPS files.
Other Resources	Files (such as sound files) for other resources used by an application.
Subprojects	Directories containing subprojects used by an application.
Supporting Files	Files not used directly by the application but that should be kept with the application.
Libraries	Libraries referenced by an application. OpenStep libraries are referenced but not copied into the project directory. Other libraries, such as those you create, may be added to the project directory.

You can use Project Builder’s file viewer to do the following:

- Browse the project and the files it contains.
- Add files to the project (as described in “Adding Files to a Project” on page 2-17).
- Remove files from the project by selecting the file in the browser and then choosing Remove in the Files menu.

- Open a project file by double-clicking on its name or icon (or, by selecting the file in the browser and then choosing Open in Workspace in the Files menu).

Adding Files to a Project

There are several ways to add an existing file to a project. The file can be already located in the project directory, or it can be somewhere else. To add it, use one of the following methods:

- Drag the file from the File Viewer into the project window. If you drag it to the suitcase in which it belongs, that suitcase opens up. If you let it go, it is added to that suitcase. If instead you drag it to the project suitcase, the project suitcase opens up and the file is added to it. The Classes suitcase takes `.m` files, the Headers takes `.h` files, and so on. Other Sources refers to files that are not headers or classes, but need to be compiled and linked into the target of the project (application, bundle or palette). Other Resources refers to files that need to be copied into the target. Supporting Files refers to files that are necessary to maintain the project, but do not become part of the target.
- Select a suitcase and choose the Add command in the Files menu (or simply double-click the suitcase). A panel is displayed in which you specify a file to add to the selected suitcase.
- Use the service that Project Builder supplies to other applications. Relevant applications have a command named Project in their Services menu. This command displays a submenu containing two commands: Add To and Build. Add To can be used to add the current file to the project (in this case, the file must already be located in the project directory).

File Display Shortcuts

The following shortcuts are available in the File display:

- Control-dragging in a file list allows you to reorder the files. This can be especially important in dealing with libraries, since the file order determines the link order.
- Alt-double-clicking on the icon of a selected file selects that file in the workspace File Viewer, instead of opening it.

- Command-double-clicking on a source file opens both the file and its associated header file, if it exists.

Building the Project

When you instruct Project Builder to build the project, the project is compiled by the `make` utility using the project's makefile. The project's source files are compiled and linked into an executable file. The project makefile provides the information the `make` utility needs to do this job. The warnings generated by the compiler and link editor provide information to help you locate and fix bugs detected at compile time.

Note – You can specify default build arguments, which apply for all projects, with the Preferences command in the Info menu. See “Setting Preferences” on page 2-29 for information on Project Builder's Preferences panel.

To build the project, first bring up the Builder display, shown in Figure 2-20, by clicking on the Builder button in the project window.

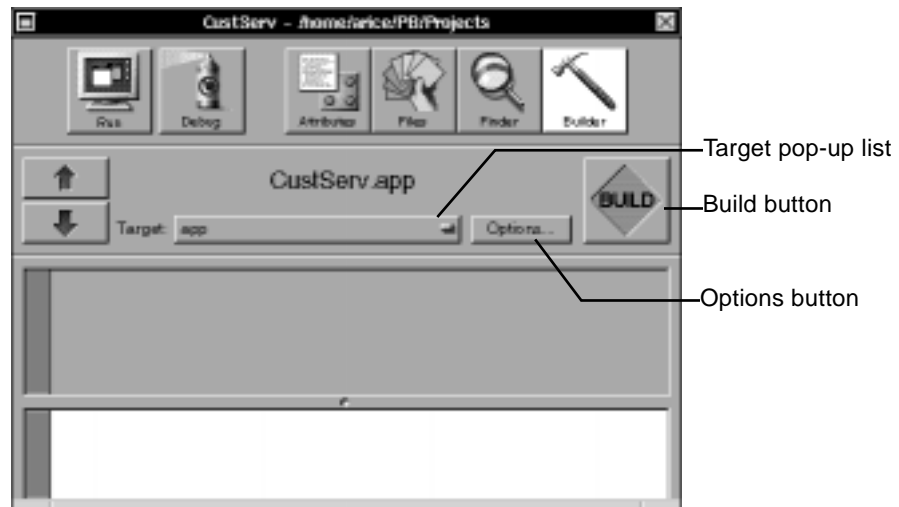


Figure 2-20 Builder Display in Project Window

The Target pop-up list, shown in Figure 2-21 on page 2-19, lets you specify a build target for the project. The build targets on the pop-up list are described in Table 2-5 on page 2-23.

You can add a custom target to the pop-up list using the Add option at the bottom of the list. When you add a custom target, it is displayed as an option in the pop-up list for the current project only.

The default build target is "app" for application projects, "bundle" for bundle projects, "palette" for palette projects, or "tool" for tool projects. You can choose another target for this project.

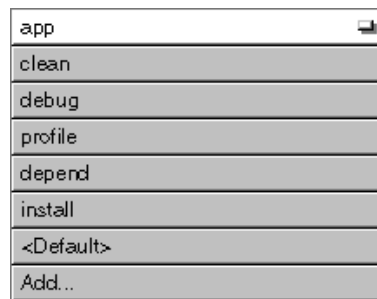


Figure 2-21 Target Pop-up List

The Options button, shown in Figure 2-22, brings up the Build Options panel (see Figure 2-23) in which you can specify the build options described in Table 2-4.



Figure 2-22 Options Button

If you want to specify the build attributes using the Build Options panel, be sure to do so before starting to build the project.

Table 2-4 Build Options

Option	Description
Arguments	Arguments to be passed on the command line to the make utility that is run during the build.
Host	The host on which the project will be built. The remote host you choose must have network access to your project directory. If you specify a host here, it overrides any host specified in the Preferences panel.
Build after error	Lets you override the Preferences setting of the option to continue building projects even when a fatal error is encountered during compiling.
Compiler	Extra command line arguments to be passed to the compiler
Linker	Extra command line arguments to be passed to the linker
Library search order	A list of directories in which the linker will search for libraries
Header file search order	A list of directories in which the compiler will search for header files

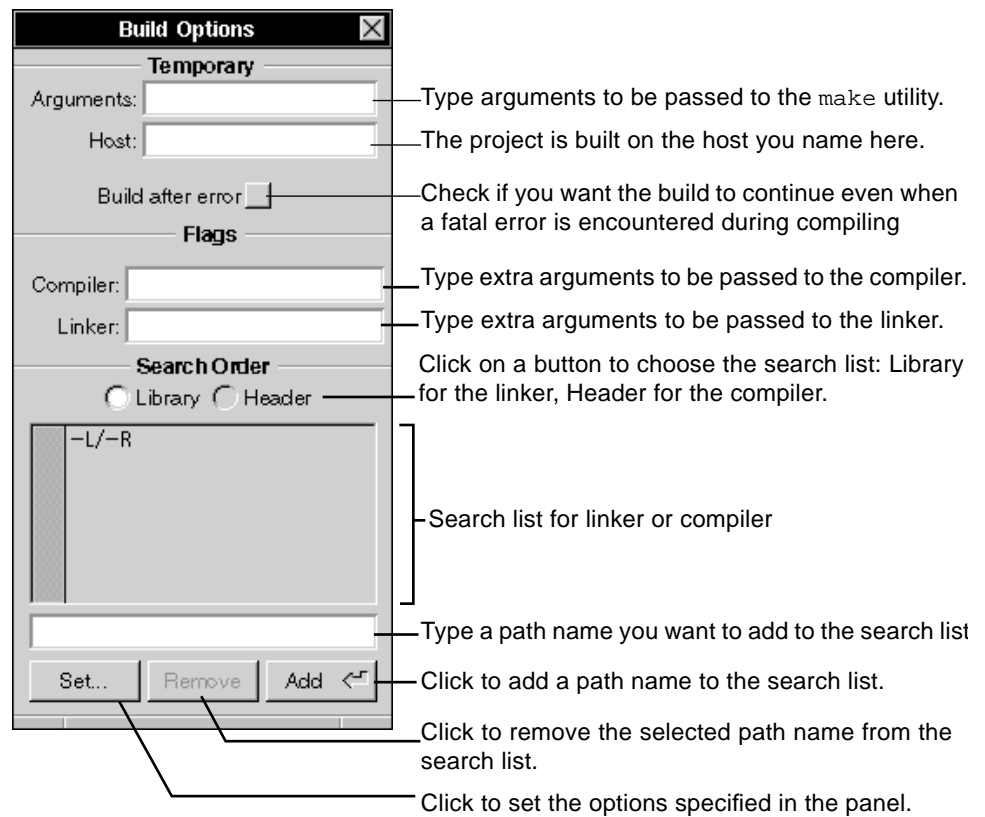


Figure 2-23 Build Options Panel

Note – If you build the project on a remote host, be sure you know what version of OpenStep the host is running.

When you are ready to build the project, click on the Build button (shown in Figure 2-24 on page 2-22).



Figure 2-24 Build Button

As the build progresses, the two views at the bottom of the window (see Figure 2-25) inform you of any warnings or error messages that occur—the upper Summary view is more selective in what it chooses to display, so you may choose to hide the lower Detail view and only refer to its output when you need to.

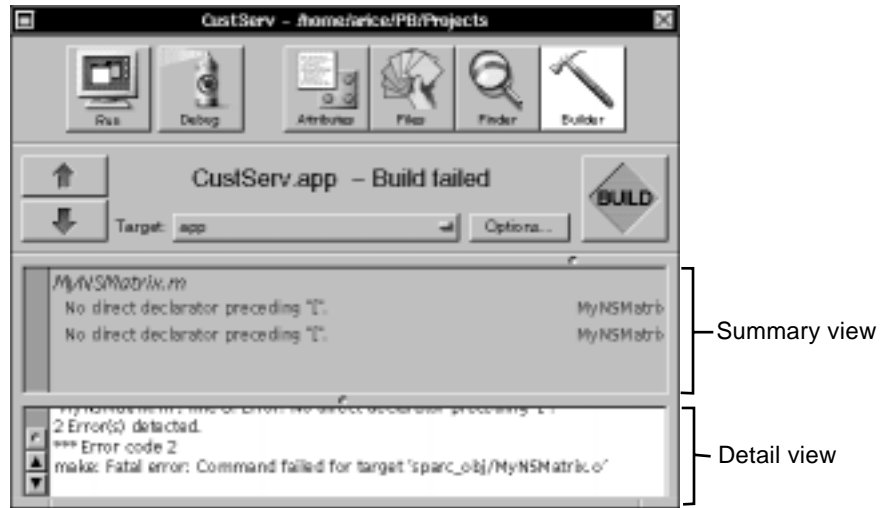


Figure 2-25 Warnings and Error Messages

If an error is encountered during the build process, a message is displayed in both the Summary view and the Detail view.

Click on a line in the Summary view to open the specified file; if you click on a line containing an error message (shown in red), the file opens in Edit and scrolls to display the line that contains the error.

Build Targets

The shared `Makefile` used to generate the executable file for all applications created with Project Builder, `app.make`, defines a number of alternate targets to perform specific tasks at various phases of the application development process. To run the `make` utility using the alternate targets, select the corresponding argument from the Targets pop-up list in the Builder display (see Figure 2-21 on page 2-19). The pop-up list provides various targets, which are listed in Table 2-5 along with the tasks they perform.

Table 2-5 Build Targets

Target ¹	Task
<code>app/bundle/palette</code>	Compiles and links an optimized version of the project. The target in this first pop-up item can be <code>app</code> , <code>bundle</code> , or <code>palette</code> , depending on the type of project you are building. The default target produces the same result.
<code>clean</code>	Removes all derived files, such as object and executable files, from the project directory, returning the project to its precompiled state.
<code>debug</code>	Compiles (with all warnings and <code>-DDEBUG</code> on) and links a debuggable, unoptimized version of the executable file with the extension <code>.debug</code> .
<code>profile</code>	Generates (with all warnings and <code>-DPROFILE</code> on) the file <code>ApplicationName.profile</code> , an executable containing code to generate a <code>gprof</code> report. This option is useful when you are performance tuning an application. See the UNIX® manual page <code>gprof</code> for details on profiling.
<code>install</code>	Compiles and links an optimized version of the project. Then copies the application into the installation directory specified in Project Builder, setting permissions and owners as appropriate, and strips the installed project. The default installation directory is <code>\$(HOME)/openstep/Apps</code> , the <code>Apps</code> directory in the user's home directory.
<code>default</code>	Compiles and links the project in the same way as the first entry in the pop-up list (<code>app</code> , <code>bundle</code> , or <code>palette</code>).

1. Project Builder does not support a “depend” build target. Instead, it uses the Solaris `make` utility's `.KEEP_STATE` mechanism (see the UNIX manual page for the `make` utility). The presence of a target name `.KEEP_STATE` in a makefile causes the `make` utility and the compiler to record dependencies in a file called `.make.state`. These dependencies are used in future builds to determine which targets are out of date.

The Preamble and Postamble Files

Every make command run by Project Builder includes a file named `Makefile.preamble` that contains definitions of all the user-configurable macros that are used in the rest of the Project Builder Makefiles. By default, this file is `/usr/openstep/Developer/Makefiles/Makefile.preamble`.

If you need to customize the make process for a project, you can create a local version of `Makefile.preamble` in the top directory of the project. The file's presence causes it to be included by the project makefiles instead of the standard `/usr/openstep/Developer/Makefiles/Makefile.preamble`.

If you need to customize the make process for a project that includes IDL interface files, see “Defining User-configurable Macros for a Project with IDL Interfaces” on page 2-27.

Your local `Makefile.preamble` should contain the following:

```
include $(OSHOME)/Developer/Makefile/Makefile.preamble
<your customizations>
```

The `include` causes the standard `Makefile.preamble` to be included. Then `<your customizations>` can override or add to any of the macros defined therein.

For example, the standard `Makefile.preamble` defines the optimization compiler option as follows:

```
OPTIMIZATION_CFLAG = -O
```

To change this to `-O2`, your local `Makefile.preamble` would contain the following:

```
include $(OSHOME)/Developer/Makefile/Makefile.preamble)
OPTIMIZATION_CFLAG = -O2
```

A `Makefile.preamble` in the top level directory of a project is included by make commands in all the directories of the project, so a customization contained therein occurs in the make command in each project directory.

You can also create a customization that applies only to a single directory of a project. To do so, create a file named `Makefile.postamble` in that directory. The presence of this file causes it to be included in the make command for that directory only.

A `Makefile.postamble` can add to or redefine any of the macros defined in `/usr/openstep/Developer/Makefiles/Makefile.preamble`. It can also define additional targets, or additions to existing targets through the `:: make` construct.

You can also build a project by entering `make` commands into a shell command line. You can specify any of the targets shown in Table 2-5 on page 2-23. For example:

```
make app install
```

builds an optimized version of the application and then installs it.

You can specify to the `make` utility, or add to the Build Targets pop-up list, the additional targets listed in Table 2-6 on page 2-26.

Table 2-6 Additional Build Targets

Target	Task
strip	Strips the optimized version by removing relocation information.
quick ¹	Runs a <code>make debug</code> command, but not a <code>make</code> command in a subdirectory unless a file in that subdirectory has been modified since the last time it was built. Use this target when you have built the project and then modified some <code>.m</code> files.
link ¹	Runs the <code>make</code> utility on only the top level directory of a project, causing modified files to be compiled and the final link to be performed.
<subdir> ¹	<p>If a project contains multiple directories, you can run the <code>make</code> utility only in the top level directory. However, from the top level, you can run the <code>make</code> utility in individual second level directories by specifying directory names as targets on the <code>make</code> command. For example, if project <code>gus</code> contains subproject <code>fred.subproj</code> and bundle <code>sam.bproj</code>, then if you modify a file in the bundle, you can quickly rebuild the project as follows:</p> <pre> : make sam.bproj </pre> <p>This runs the <code>make</code> utility in the bundle but will not rebuild the top level directory or the subproject. If you modify a file in the subproject, then you could use the following:</p> <pre> make fred.subproj link </pre> <p>The <code>link</code> target is necessary since the result of running the <code>make</code> utility in a subproject is a <code>.o</code> file that must be linked into the top-level executable. The <code>link</code> target is not necessary on the <code>make sam.bproj</code> command since bundles are dynamically loaded and do not have to be linked into the top-level executable.</p>

1. These targets are intended to speed up the execution of `make` commands during the debugging process. Each will run a debug mode `make` command in portions of a multi-directory project. You should use these targets only when you are sure that the partial updates that result will suffice.

Defining User-configurable Macros for a Project with IDL Interfaces

If you have included IDL interface files in your project using Interface Builder (as described in “Adding IDL Template Objects to Your Interface” on page 3-164), you must create a local version of `Makefile.preamble` in the top directory of the project as described in “Creating a `Makefile.preamble`”, or perform the steps described in “Using Project Builder to Define User-configurable Macros” on page 2-28.

Creating a `Makefile.preamble`

The `Makefile.preamble` file for a project that includes IDL interface files must contain the following:

- Modifications to the `COMMON_CFLAGS` macro to set include file paths for the header files generated from IDL interface files included using Interface Builder, if the headers are in different directories than the IDL interface files. If the headers are in the same directories, then Project Builder sets up the paths.
- Modifications to the `OTHER_LDFLAGS` macro to include paths and shared libraries for the IDL stub libraries generated from the IDL interface files.
- Modifications to the `OTHER_LDFLAGS` macro for the path and name of the ODF library.

If the server with which you intend to connect is running in development mode, you also need to modify `OTHER_LDFLAGS` to include an additional relocatable, `/opt/SUNWdoe/lib/odf/development_mode.o`. This is required for successful name service lookup on NEO servers running in development mode.

To find out if the server is running under the development mode start `neoadmin` from a shell command line. In `neoadmin`, type the `server` command to view the NEO servers. If the service of interest has a `.development` extension, it is running in development mode. For more information on how to use `neoadmin`, see .

The following is a sample `Makefile.preamble`:

```
-----  
include $(PREAMBLE)  
COMMON_CFLAGS =-I/opt/SUNWdoe/include \  
               -I<include path>/odf_output  
OTHER_LDFLAGS =-L/opt/SUNWdoe/lib \  
               -R/opt/SUNWdoe/lib \  
               -L<library path> \  
               -R<library path> \  
               -lOdf -l<stub library>  
OTHER_OFILES += /opt/SUNWdoe/lib/odf/development_mode.o  
-----
```

Using Project Builder to Define User-configurable Macros

Instead of creating a `Makefile.preamble`, you can define the user-configurable macros for a project that includes IDL interface files by using the project window and performing the following steps:

- 1. Click on the Builder button in the Project Window to bring up the Builder display. Click on the Options button to open the Options panel.**
- 2. Add the `-I` options to the Header Search Order list, and add the `-L/-R` options to the Library Search Order list.**
- 3. Click on the Files button in the Project Window to bring up the Files display.**
- 4. Drag the Odf library name and the `<stub library>` name from the FileViewer to the Libraries suitcase in the Files display.**
- 5. Drag the `development_mode.o` file from the File Viewer to the Other Sources suitcase. Unfortunately, this will make a copy of it. If you do not want a copy, then before dragging the file, use an `ln -s` command to link this `.o` file into the top directory of the project. When you drag the file to the Other Sources suitcase, an attention panel will tell you the file already exists and ask if you want to replace it. Reply No.**

Setting Preferences

You can specify preferences for a variety of options using the Preferences panel. To bring up the panel, choose the Preferences command in the Info menu.

Enter values or click on buttons to specify new preferences. Then click on Set to set the new preferences (or click on Revert to restore the previous settings).

Note – The settings on the Preferences panel are global—they apply to all projects, not just the current project. The settings can be overridden for a specific project by settings you specify in the Build Options panel for that project (see Figure 2-23 on page 2-21).

Build Defaults Controls

The controls in the Build Defaults group, shown in Figure 2-26, let you specify alternate targets to display in the Builder display's Targets pop-up list (see Figure 2-21 on page 2-19). They also let you specify build arguments to be passed to the `make` utility and a remote host on which to build the project. The switch lets you choose to continue building a project even when a fatal error is encountered during compiling.

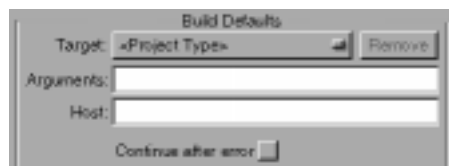


Figure 2-26 Build Defaults Controls

Tools Controls

The controls in the Tools group, shown in Figure 2-27 on page 2-30, let you specify the programs to use to edit source code and debug the executable—these files are used in interactive debugging with Project Builder. You can also specify an alternative to `/bin/make`, the standard `make` utility.



Figure 2-27 Tools Controls

The default editor that Project Builder uses to display files is the OpenStep Edit application. However, you can configure Project Builder to use a different editor by entering a command to invoke that editor in the Editor field of the Preferences panel. If the Editor field does not contain a command that invokes an OpenStep application, then Project Builder does the following when it opens a file in the specified editor:

- If the command in the field is `vi`, Project Builder runs the following command:

```
/usr/openwin/bin/xterm -e vi +<lineNum> <fileName>
```

- If the command begins with `xemacs`, Project Builder runs the following command:

```
gnuclient -q +<lineNum> <fileName>
```

In order for this command to work, the following conditions must exist:

- The `gnuclient` application must be found in a directory in your search path. It is normally in same directory as the `xemacs` executable file.
- You must already have the `xemacs` application running and you must have run the `xemacs` command `M-x gnuserver-start`. This command starts a daemon that is a go-between between the `xemacs` and `gnuclient` applications.
- If the Editor field contains any command other than `vi` or `xemacs`, Project Builder runs the following

```
<command> +<lineNum> <fileName>
```

For example, if you want to use `shellTool` instead of `xterm` to run the `vi` application, the command could be the following:

```
/usr/openwin/bin/shelltool vi
```

Sounds Controls

The controls in the Sounds group, shown in Figure 2-28, let you specify the sound cues for Project Builder to use when the project builds successfully, and when the project fails to build.

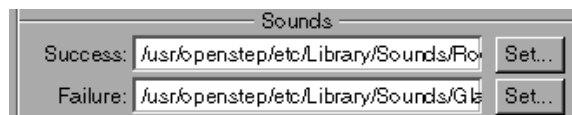


Figure 2-28 Sounds Controls

Build Service Controls

The controls in the Build Service group, shown in Figure 2-29, let you specify what (if anything) you want to have happen after building your project (specifically, after building your project by choosing Project Builder's Build command on the Services menu)—Build only, Build and Run, or Build and Debug.

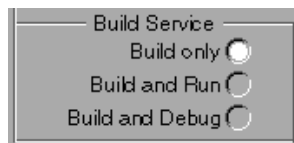


Figure 2-29 Build Service Controls

Save Options Controls

The controls in the Save Options group, shown in Figure 2-30, let you specify whether projects should be autosaved, and whether the most recent backup file is automatically deleted or retained.

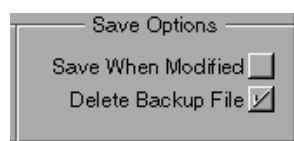


Figure 2-30 Save Options Controls

Running and Debugging an Application

In addition to maintaining and building a project, you can use Project Builder to run or debug the resulting application, as described in the following sections.

Running

To run the project application, click on the Run button in the project window, shown in Figure 2-31. If the project has not been built yet, it is built and then the application is run. The Run button's icon is the same as the application icon—the icon shown here is the default application icon that Project Builder uses if no other icon is specified in the Attributes display.

Alt-clicking on the Run button runs the application without building it first.



Figure 2-31 Run Button

Debugging

To debug the project application, click on the Debug button in the project window, shown in Figure 2-32. If the project has not been built yet, it is built first and then the application is run in debug mode.

Alt-clicking on the Debug button runs the application under the SPARCworks Debugger without building it first.



Figure 2-32 Debug Button

When you indicate that you want to debug an application in Project Builder, the following steps occur:

- The project is built (unless it is already up to date).
- Terminal creates a new window in which to run the Debugger process.

For information about using the SPARCworks Debugger, see the SPARCworks manual *Debugging a Program*.

Project Builder Command Reference

Project Builder's main menu contains the standard Info, Edit, Windows, Services, Hide, and Quit commands. All commands unique to Project Builder are located in the Project and Files submenus—these menus and the commands they contain are described below.

Commands in the Project Menu

The Project menu contains commands for creating and maintaining your projects, as listed in Table 2-7.

Table 2-7 Project Menu Commands

Command	Description
New	Creates a new project.
Open	Opens an existing project.
Open Makefile	Opens a window for just the Makefile of a project and displays the Builder view in the window. To build the project, click on the Build button.
Save	Saves the current project.
New Subproject	Creates a new subproject
Add Help Directory	Adds a Help directory to the current project. A template Table of Contents file and Index file are placed in the Help directory. For more information on adding help to an application, see "Attaching Help to Objects" on page 3-132 and "Adding Help Links" on page 4-17.

Table 2-7 Project Menu Commands (Continued)

Command	Description
Run Application	Runs the application associated with the project, just as if you had clicked on the Run button in the project window.
Debug Application	Debugs the application associated with the project, just as if you had clicked on the Debug button in the project window.
Build Application	Builds the application associated with the project, just as if you had clicked on the Build button in the project window.

Commands in the Files Menu

The Files menu contains commands, shown in Table 2-8, that affect the files that make up a particular project. Commands in this menu are enabled only when the Files view for the project is selected.

Table 2-8 File Menu Commands

Command	Description
Add	Adds a file to the selected suitcase in the current project. Be sure to select the appropriate suitcase in the Files display before choosing the command.
Open in Workspace	Opens the selected file in the application that is registered with the Workspace Manager as the default application for files of that type.
Select in Workspace	Displays and highlights the selected file in the Workspace Manager's File Viewer window.
Remove	Removes the selected file from the current project (without deleting it from the project directory).
Sort	Alphabetically sorts the files in the current suitcase.
Make Global	Makes the selected file global (that is, moves it from the <code>Language.lproj</code> directory into the project directory).

Table 2-8 File Menu Commands (Continued)

Command	Description
Make Localizable	Makes the selected file localizable (that is, moves it from the project directory into the <i>Language.lproj</i> directory).
Make Public	Makes the selected header file public. The file is flagged in the File view in the Project window. Public files are also flagged in the <i>PB.Project</i> directory and the makefile so that they are installed in a target directory where they can be accessed by others when the header files are installed by the <i>Make</i> utility.
Make Private	Makes the selected public header files private; that is, reverses the effects of <i>Make Public</i> .

Working with Interface Builder



Interface Builder is a tool that helps you design and build applications. It speeds the creation of applications by letting you define an interface (and in some cases, an entire application) graphically rather than by writing C and Objective C code. With Interface Builder, you drag objects from palettes of OpenStep objects directly into the application you are building. Once there, an object can be modified in ways that are specific to its class: You can set an `NSButton` object's title or set the minimum and maximum values of an `NSSlider` object, for example. After you have gathered and edited the objects that will make up your application, Interface Builder lets you define the interactions among them and associate help messages with each of them. Even before you write a line of code, you can run your application within Interface Builder to check the operation of its interface.

Interface Builder's technique of direct manipulation of programming objects is not limited to objects defined in OpenStep. Interface Builder's palettes are extensible, letting you load palettes containing objects that you or other developers have created.

In many ways, using Interface Builder to create an application is much like using a graphics editor to create a drawing. However, Interface Builder is not a simple "screen painter" or form-generation tool. When you build an application with Interface Builder, you are interacting with the actual programming code that will be run when your application runs on its own. The objects you manipulate in Interface Builder are the objects that will appear in the working version of your application. If your application runs correctly in Interface Builder, it will run correctly on its own.

The work you do in Interface Builder is saved in a nib file (a file package having a name ending in `.nib`). This file contains archived versions of the objects you assembled for your application, information about connections between these objects, and other information. When an application begins running, it unarchives these objects and associated information from one or more nib files. Projects in OpenStep contain at least one nib file and Interface Builder lets you create and modify these nib files.

The central tool for developing applications in OpenStep is Project Builder. When you start a new project in Project Builder, you are provided with several standard components, one being a nib file. When you want to modify this standard nib file, Project Builder invokes Interface Builder as the nib file's editor. Interface Builder and Project Builder are interlinked in other ways as well. As you define new classes, import images or sounds, or create new nib files, Interface Builder and Project Builder work together to keep each other aware of the state of the project.

Even if you are new to this computing environment, you will find that with Project Builder and Interface Builder, you will be able to create applications with a minimum of time and effort. This efficiency results from working directly with the application's objects, rather than with files of programming code. However, the more you know about the Application Kit and the more comfortable you are with programming in the Objective C language, the easier application development will be for you. Thus, we recommend that you familiarize yourself with the material in Chapter 8, "The Objective C Language" and Chapter 7, "The NSObject Class," and at least scan the class specifications (located in Chapter 1 of *OpenStep Programming Reference*) for the major Application Kit classes before attempting to take your work with these tools beyond the experimental stage.

This chapter provides both general reference information and detailed task-oriented information on Interface Builder. It first introduces Interface Builder's major components and then discusses the tasks that you use Interface Builder to accomplish. A final section provides a quick reference for each of Interface Builder's commands.

Interface Builder's application programming interface (API), which allows you to create custom palettes, is described in detail in Appendix B, "Interface Builder Application Programming Interface," Appendix C, "Interface Builder API Classes," Appendix D, "Interface Builder API Protocols," and Appendix E, "Interface Builder API Types and Constants."

An Orientation

When you use Interface Builder, its windows—and the windows of the application under construction—share the screen. Figure 3-1 gives you an idea how this looks.

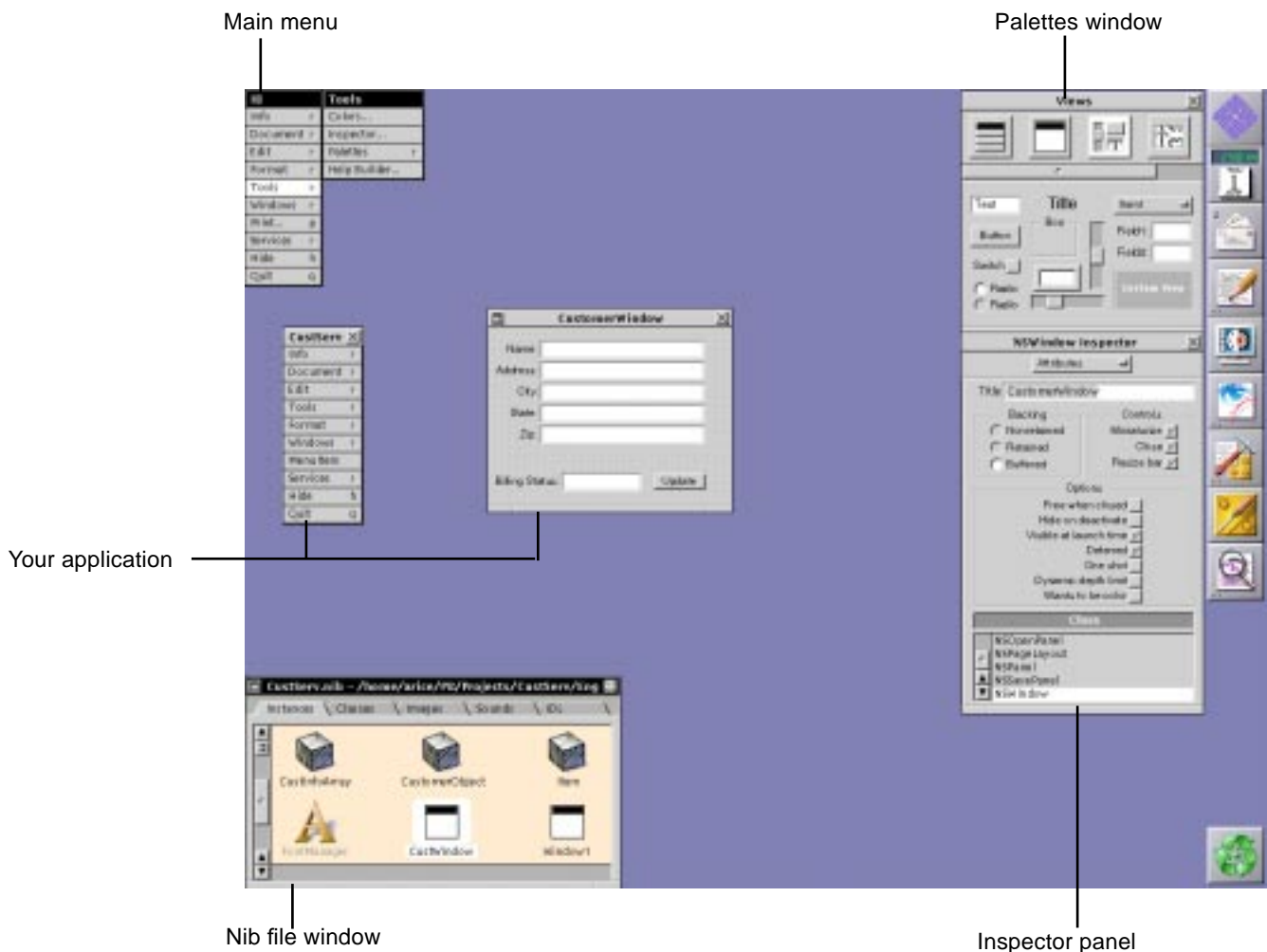


Figure 3-1 Interface Builder and Your Application

Interface Builder's windows frame an area of the workspace where you build your application. At the upper left is the main menu, which gives you access to Interface Builder's tools and commands, and at the upper right is the Palettes window.

The *Palettes window* is the source of objects (NSButtons, NSSliders, NSWindows, and so on) that you can drag into your application. The Palettes window is described in detail in “Using the Palettes” on page 3-27.

Below the Palettes window is the *Inspector panel*. You use this panel to set the attributes of an object, to connect it to other objects, and to review the attachments between objects and help messages. The Inspector panel is described in “The Inspector Panel” on page 3-21.

At the bottom left is the *nib file window*. The nib file window displays your application's top-level objects (its windows, main menu, and so on) and gives you access to the image, sound, and class resources that are available to your application. For a description of the nib file window and its various displays, see “The Nib File Window” on page 3-10.

Building an Application with Interface Builder

The Application Kit defines a library of user-interface objects that you can select from for your application. Interface Builder makes the selection process a graphical one: You simply drag the object you need from the Palettes window into the application you are building. By building an application in this way, you can be sure that its interface will work properly and will, in a broad sense, conform to the interface standards for OpenStep applications. See “Using the Palettes” on page 3-27 for a description of the Palettes window and the objects available to you on the OpenStep palettes.

Specifying Object Attributes

Once an object is added to your application, you can adjust the values of many of its instance variables directly. For example, to change the size of a button, you drag one or more of its sides to a new position. Changing the image on the screen changes the value of the NSButton object's *frame* instance variable. For attributes that are not easily represented graphically, Interface Builder provides the Inspector panel that lets you set the values for particular instance variables.

You set the maximum and minimum values of a slider with the `NSSlider Size` inspector, for example. The Inspector panel is described in “The Inspector Panel” on page 3-21.

Interconnecting Objects

Interface Builder also lets you interconnect objects so that they can communicate with one another. For example, a button can be connected to the window it is displayed in so that when the button is clicked on, the window closes. Such connections are made through an object’s outlets and actions.

An *outlet* is an instance variable that identifies another object in the application. Common examples of outlets include an `NSControl`’s *target* or an `NSApplication` or `NSWindow` object’s *delegate*. An `NSTextField` object may have an instance variable `nextText` that points to another `NSTextField` object.

An *action* is a message that one object sends to another when it receives a certain message of its own. For example, an `NSButton` object sends an action messages when it receives the `performClick:` message (when the user clicks on the button).

Adding Code to Your Application

The objects in the Application Kit are general-purpose and fill the needs of a wide cross-section of applications. What makes your application unique is the code you write. For example, the Application Kit provides the `NSButton` objects and other `NSView` objects you need to implement an interface for a calculator, but you have to create the computational engine. Interface Builder helps you declare classes that encapsulate the code that is unique to your application.

Interface Builder and the Objective C language encourage a style of programming that puts your application’s unique code in one or more objects of your own design. The application’s user-interface objects handle routine business, such as displaying the main menu or hiding the application, and also serve to interpret the user’s actions for the objects you design. If the user clicks on the calculator’s Add button, the Application Kit highlights the button and then sends a message to your calculator object to perform the addition.

Using this style of programming, your application will generally contain a number of standard Application Kit objects and one or more subclasses of `NSObject` and `NSView`. Most often, the subclasses of `NSObject` embody the logic that is unique to your application, and the `NSView` subclasses contain the drawing code that is unique to your application. You will rarely need to create subclasses of other Application Kit classes.

Composing the Interface

To compose your application's interface, you just drag objects off a palette, drop them on a window or other "surface," and then manipulate and arrange them into an effective user interface. This section introduces you to Interface Builder by showing how to compose the elements of your interface.

Opening a Nib File

To open your application's nib file, do one of the following:

- Double-click a nib file in Project Builder.

Or

- Double-click a nib file in the nib file window.

Or

- In Interface Builder, choose the Open command and select a file in the Open Panel.

You will usually open a nib file in Project Builder, as shown in Figure 3-2, since that is the central tool for application development. When you create an application in Project Builder, an empty nib file is automatically created for you and added to the project's Interfaces suitcase. This file has the same name as your application project and, like all nib files, ends with the extension `.nib`.

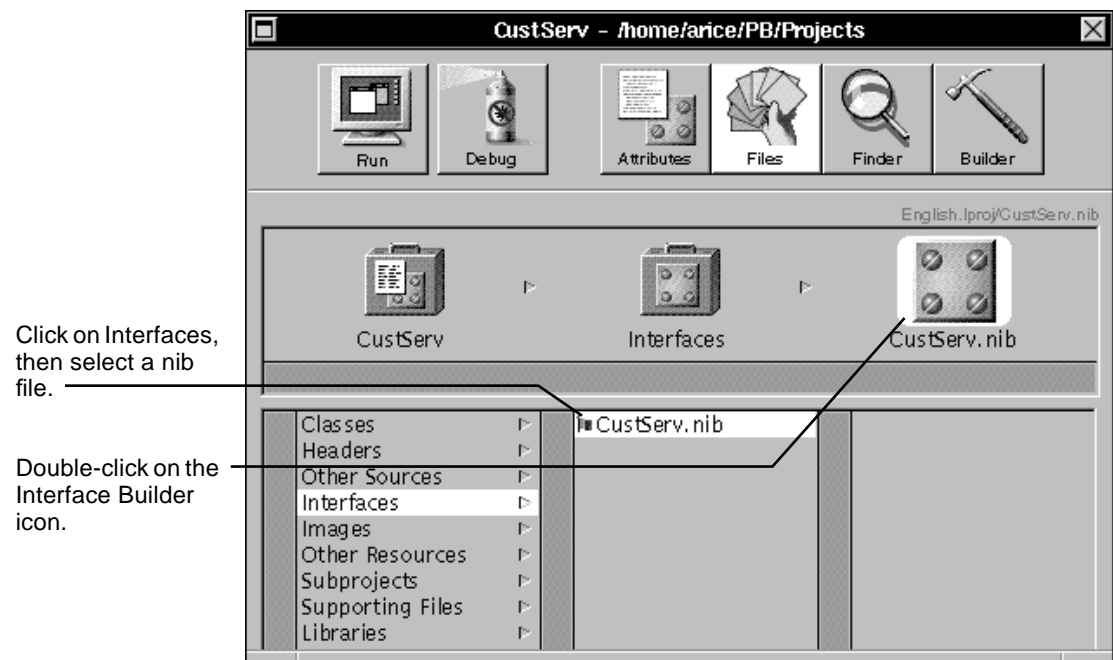


Figure 3-2 Opening a Nib File in the Project Builder Window

You can also open nib files directly in the Workspace's File Viewer by double-clicking on them. And you can open nib files from within Interface Builder by choosing the Open command from the Document menu; in the Open panel locate and select the file as shown in Figure 3-3.

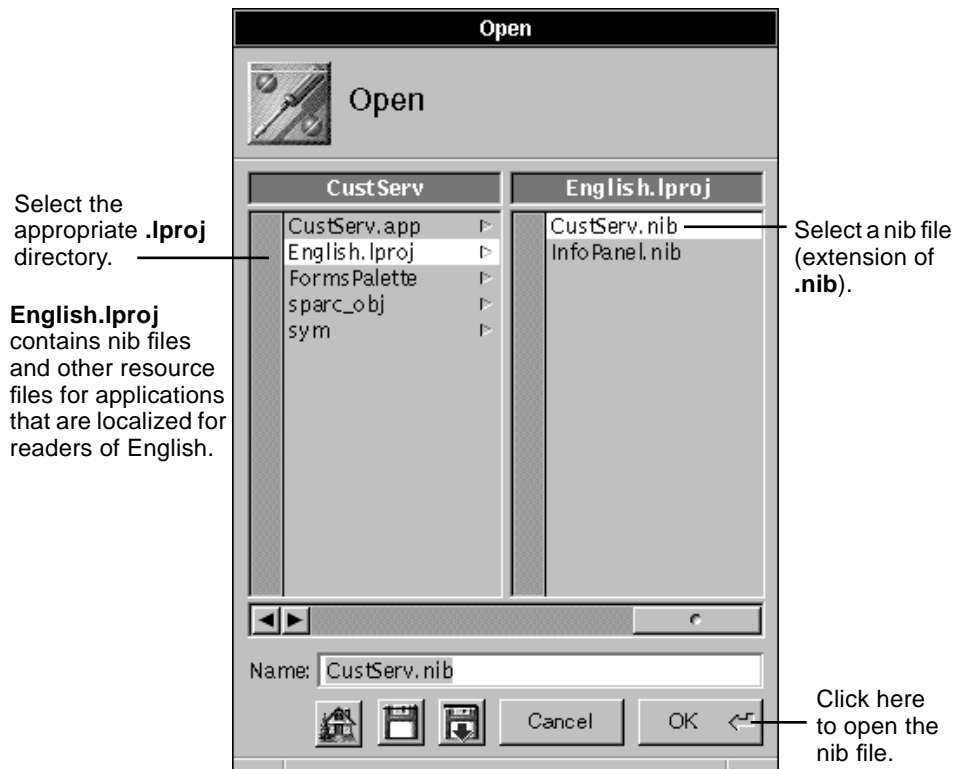


Figure 3-3 Opening a Nib File in the Open Panel

Nib files (so called because of their `.nib` extension) archive the class definitions, objects, and the connections between objects when you create an interface in Interface Builder. See “What Is in a Nib File” on page 3-24 for some conceptual background on nib files.

When Interface Builder Starts

When you open a nib file, Interface Builder displays several windows and panels on your screen.

The Palette Window

The palette window, shown in Figure 3-4, holds all currently loaded palettes of objects. You select a palette by clicking on its icon (if it is not already visible). Then drag objects from the palette to the appropriate surface.

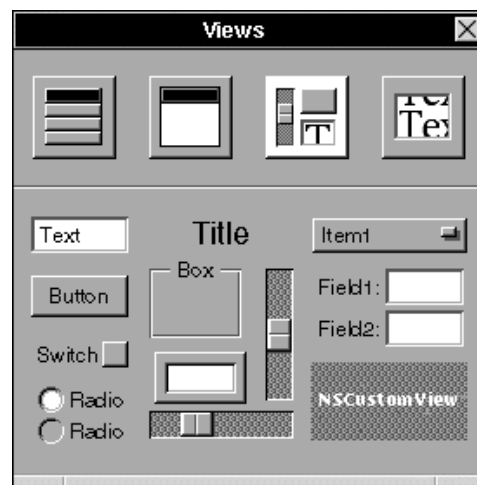


Figure 3-4 The Palette Window

The Application Kit palettes as described in detail in “Using the Palettes” on page 3-27.

The Interface Window

The interface window or panel displays the actual interface on which you are working. If this is the first time you have opened a main nib file in Project Builder, an empty window is displayed. Figure 3-5 shows an example of an interface window.

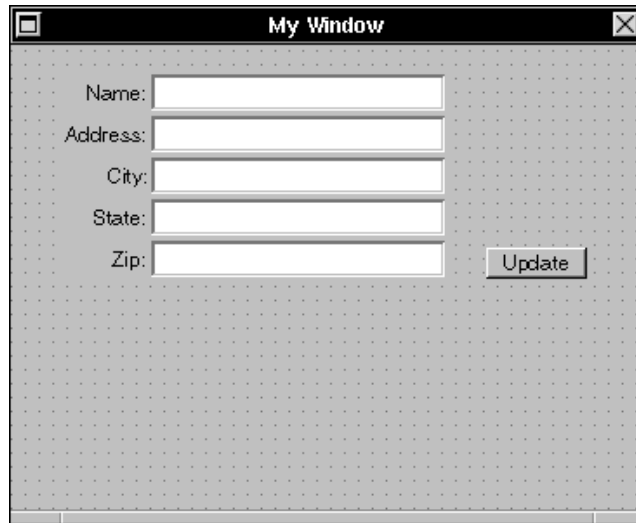


Figure 3-5 An Interface Window

The Nib File Window

The nib file window, shown in Figure 3-6, contains multiple views that display the contents of the nib file. Selected by clicking on a folder tabs, these views show archived objects; the connections among objects, the current class hierarchy (including any custom classes that you may have created), and the images and sounds stored in the nib file.

Each window in the nib file is represented by a window icon in the nib file window. By double-clicking on a window icon, you can bring the window it represents to the front so that the objects it contains are visible. The nib file window also contains icons that represent the file's owner object and a first responder object, objects that are discussed in "The Nib File's Owner" on page 3-18 and "The First Responder Object" on page 3-20.

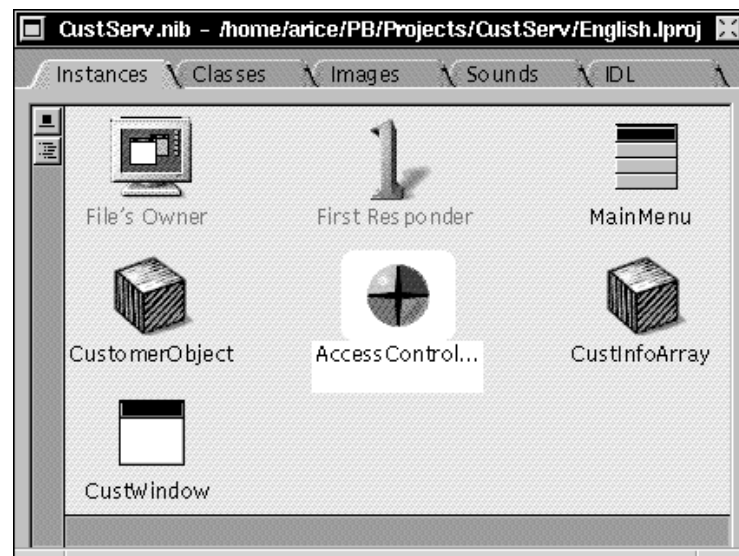


Figure 3-6 The Nib File Window

The nib file window contains five displays:

- Instances display in icon mode
- Instances display in outline mode
- Classes display
- Images display
- Sounds display
- IDL display

Icon Mode of Instances Display

To display the object of your application's interface as icons, use the icon mode of the nib file window's Instances display, shown in Figure 3-7.

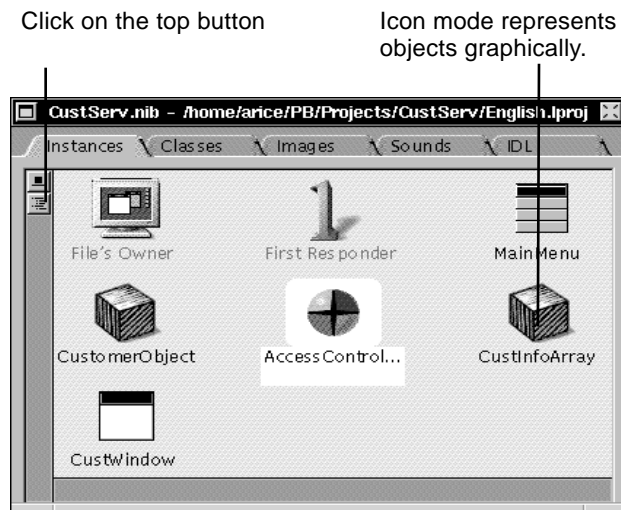


Figure 3-7 Icon Mode of Instances Display

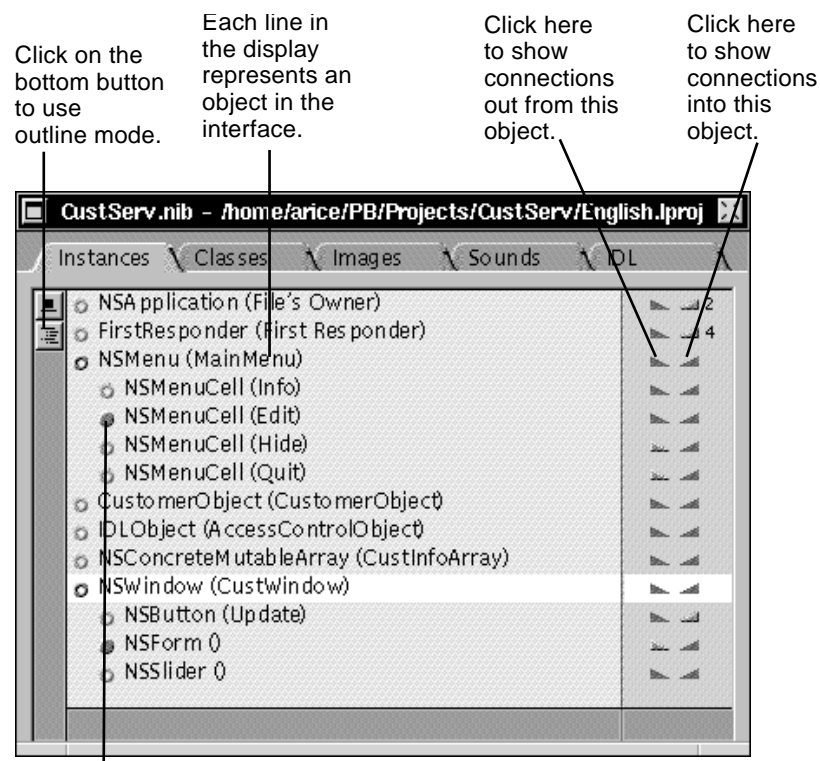
Only the top level of your interface's object hierarchy is displayed in icon mode. *Object hierarchy* is the hierarchy of objects in your interface. For example, any button on a window is displayed below that window in the object hierarchy. Icon mode would display only the window, not the button.

To display all of the objects in the object hierarchy, use the outline mode of the Instances display (see “Outline Mode of Instances Display” on page 3-12).

To select an object in icon mode, click on it. To display an object in the workspace, double-click on it.

Outline Mode of Instances Display

For a view of all of the objects in your application's interface, use the outline mode of the nib file window's Instances display shown in Figure 3-8.



If this button is filled, the subhierarchy below this object is hidden.

Figure 3-8 Outline Mode of Instances Display

Outline mode displays the entire object hierarchy of your interface. For example, any button in a window is displayed below that window in the object hierarchy. Outline mode shows the button indented below the window that contains it.

Click on the button to the left of an object's name to see the objects that this object owns. Click on the button again to hide these objects. If the button is empty and shaded, the object it represents does not own any objects.

The buttons to the right of each object show connections to and from that object. Click on the left button to see the object's outlets and the action messages sent to the object. Click on the right button to see which objects have outlets into the object and the action messages the object sends to other objects.

To select an object in outline mode, click on it. To display an object in the workspace, double-click on it.

Classes Display

Use the Classes display of the nib file window, shown in Figure 3-9, to do the following:

- Subclass OpenStep classes
- Add already-defined subclasses to the nib file
- Change the definition of a class
- Delete classes

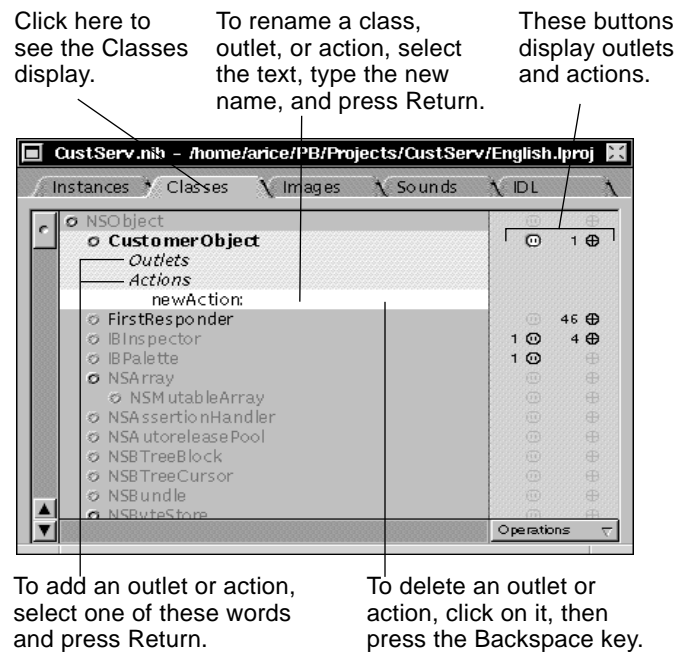


Figure 3-9 Classes Display

See “Creating a Class” on page 3-137 for information on using this display to add, modify, and delete subclasses.

The Classes display shows the classes known to your interface. Black titles are used for classes you have added to the interface. These are the only classes you can modify and delete. Gray titles are used for OpenStep classes.

Images Display

Use the Images display of the nib file window, shown in Figure 3-10, to add images to your interface and to add an image to an object in your interface. The images display shows the images known to your interface.

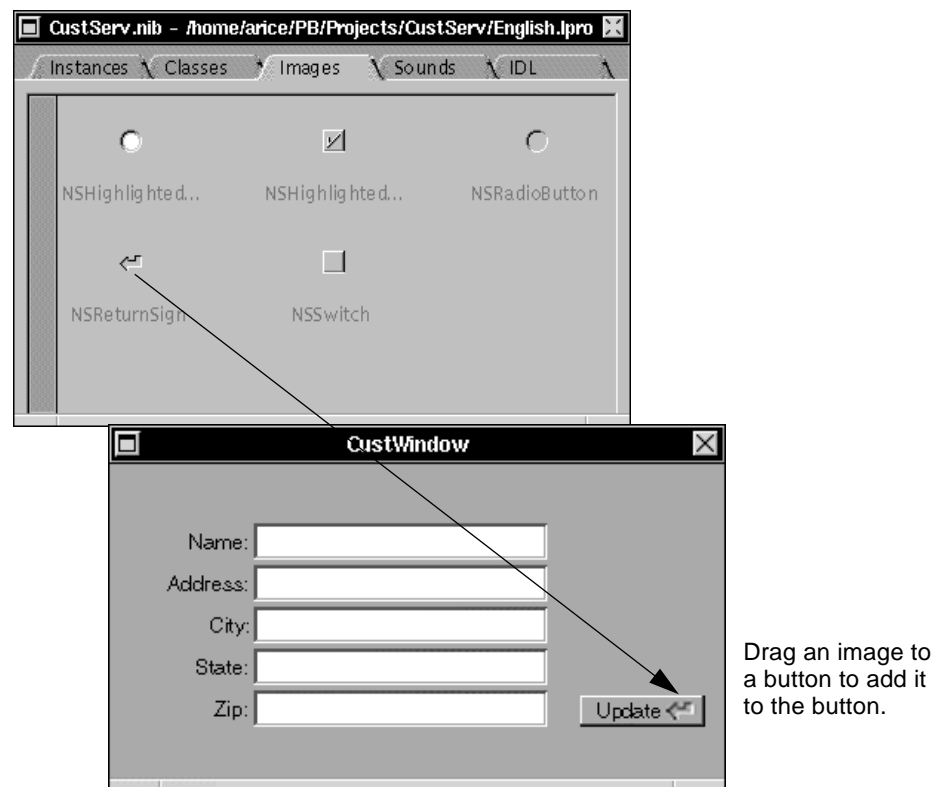


Figure 3-10 Images Display

A new nib file contains images for radio buttons, switches, and the return symbol, which you can add to a button to have it perform a click when the user presses the Return key.

Black titles are used for images local to the nib file. You can rename these images, and you can delete an image by selecting it and pressing the Backspace key. Gray titles are used for standard images and images added to the project using Project Builder.

To see what an image looks like before you add it to your interface, select it, then choose Tools from the Interface Builder menu, and choose Inspector from the Tools menu. For information on the attributes of an image that are displayed in the Inspector window, see “Managing Sounds and Images” on page 3-78.

Note – You can also add an image to your interface by dragging a TIFF or EPS file from the File Viewer.

Sounds Display

Use the Sounds display of the nib file window, shown in Figure 3-11, to add sounds to your interface and to add a sound to an object in your interface. The Sounds display shows the sounds known to your interface.

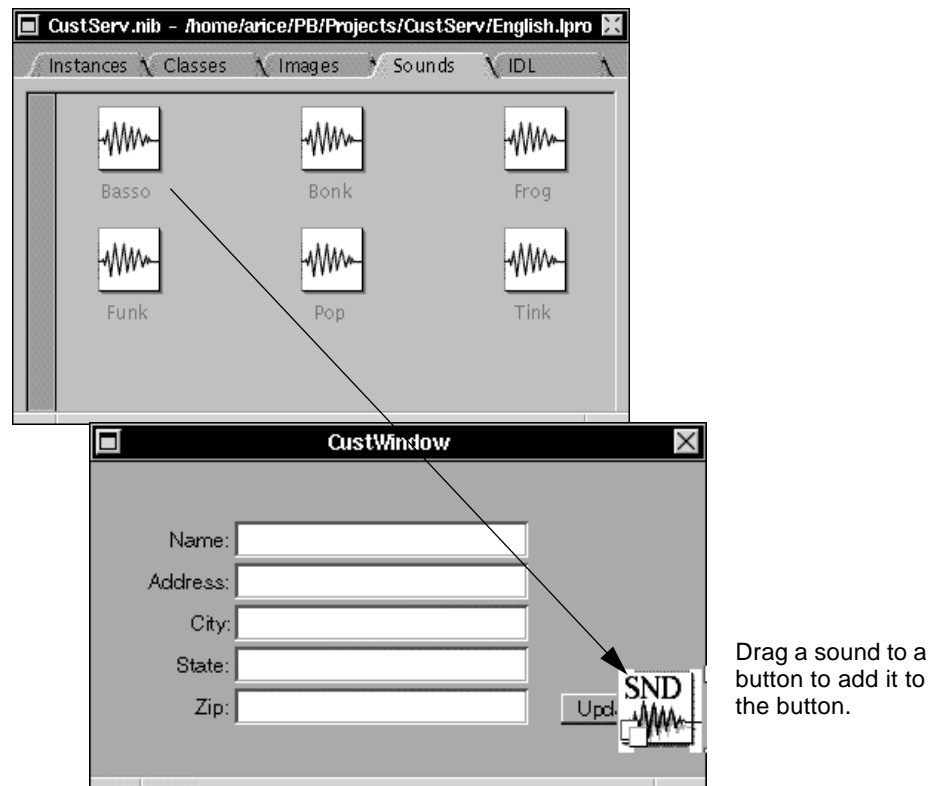


Figure 3-11 Sounds Display

All nib files contain a subset of the sounds in the standard sounds directory, `/usr/openstep/Library/Sounds`. Also, all sounds you add to the project using Project Builder are added to the nib file.

Black titles are used for sounds local to the nib file. You can delete a sound by selecting it and choosing `Cut` from the `Edit` menu. Gray titles are used for standard sounds and sounds added to the project using Project Builder.

To hear a sound before you add it to your interface, select it, choose `Tools` from the `Interface Builder` menu, then choose `Inspector` from the `Tools` menu. For information on the attributes of a sound that are displayed in the `Inspector` window, see “Managing Sounds and Images” on page 3-78.

Note – You can also add a sound to your interface by dragging a file with the extension `.snd` or `.au` from the File Viewer.

IDL Display

Use the IDL display of the nib file window, shown in Figure 3-12, to do the following:

- Parse IDL (Interface Definition Language) interface files
- Instantiate IDL template objects in the nib file

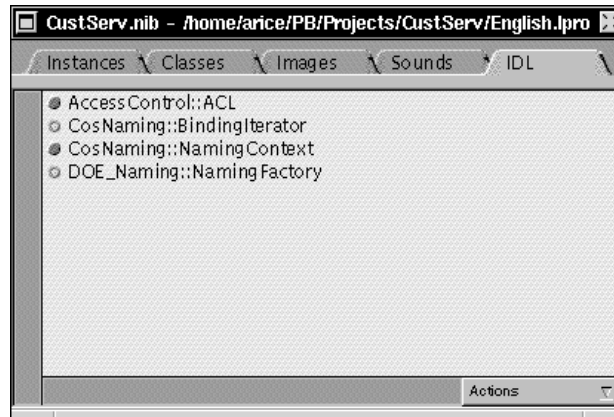


Figure 3-12 IDL Display

The IDL display shows the IDL types known to your interface. You can instantiate template objects from these IDL interfaces (see “Adding IDL Template Objects to Your Interface” on page 3-164). Then you can connect these IDL objects to outlets in instances of custom classes you have defined (see “Connecting Objects” on page 3-113).

The Nib File’s Owner

The nib file’s owner is an object that is external to the nib file and that is the conduit for messages between the objects that will be unarchived from the nib file at run time and the other objects in your application. In general, the core objects in your application access the objects unarchived from the nib file

indirectly through the owner object. In turn, the unarchived objects communicate with the other objects in your application by sending messages to the owner object.

Each nib file has one—and only one—owner. For small applications, the owner is generally `NSApp`, the application object itself, although it can be an object of any class. You can change the type of object that owns an auxiliary nib file using the Inspector window.

The owner is the only external object that may be the explicit target of action messages from `NSControls` within the nib file. The owner may also have outlets that will be initialized at run time to point to the objects within the nib file.

The owner of a nib file is represented by an icon in the nib file window, shown in Figure 3-13. You use this icon to make connections between objects stored in separate nib files. When the application object is the file's owner, the icon is a terminal.



Figure 3-13 File's Owner Icon

The owner must exist before the interface objects are loaded. For example, Project Builder generates a main file that follows this sequence of messaging to create the owner, load the interface information, and then run the following:

```
NSApplication *app = [NSApplication sharedApplication];
if ([NSBundle loadNibNamed: @"CustServ.nib" owner: app])
    [app run];
```

Note – `NSApp` is a global variable that identifies the `NSApplication` object, the object that is created by the message in the first line of the example above. (For more information on the `loadNibSection:owner:withNames:` method—and especially on the search path it uses for locating the appropriate nib file to load—see the specification for the `NSApplication` class.)

What happens when the nib file is loaded at run time is described in “When You Load a Nib File” on page 3-27.

The First Responder Object

The First Responder icon in the nib file window, shown in Figure 3-14, represents the object within a window that will be the first to receive keyboard events, mouse-moved events, and action messages from `NSControl` objects that do not have an explicit target.



Figure 3-14 First Responder Icon

In most cases, a window's first responder is either one of its `NSText` objects or one of the objects that use `NSText` objects (such as `NSForm`, `NSTextField`, and `NSScrollView` objects). Clicking on one of these objects generally makes it that window's first responder.

Over time, many different objects can become the first responder, but at any one time, only one object has this status. The First Responder icon stands for the object that has this status, no matter which actual object it is within your application. In this respect, the First Responder icon is really a fiction since it identifies no one particular object, but rather any object having a particular status. This fiction, however, is very useful.

The object represented by the icon changes when the position of the text pointer changes. For example, if you click on the pointer in a text field labeled `Name` and start to type, the `Name` text field receives the keystrokes. It is the first responder. If you move the pointer to a field labeled `Address` and start to type, `Address` receives the keystrokes, so it is the first responder.

You can use the First Responder icon to send messages to any object that currently contains the text pointer. For example, the `Cut`, `Copy`, and `Paste` commands in the `Edit` menu send messages to the first responder object.

Having First Responder in the nib file window lets you connect an object, such as the `NSMenuItem` that sends the `copy:` message, so that it sends its action message to a target whose identity changes over time. Thus, for example, the `Copy` command can be set up to work with any `NSTextField` in a window, as long as the `NSTextField` is the first responder. If you create a new application

in Project Builder, open its nib file, and check the connections in the application's Edit menu, you will discover that all Edit commands are connected to the First Responder.

The ability to let the target of a message be defined at run time rather than at compile time is an example of dynamic binding in Objective C. For more information on Objective C, see Chapter 8, "The Objective C Language," and Chapter 9, "The Objective C Extensions.")

The Inspector Panel

The Inspector panel, shown in Figure 3-15, is a multiform panel that displays the attributes, connections, and size of a selected object. It also presents the object's resizing characteristics and its associated help.

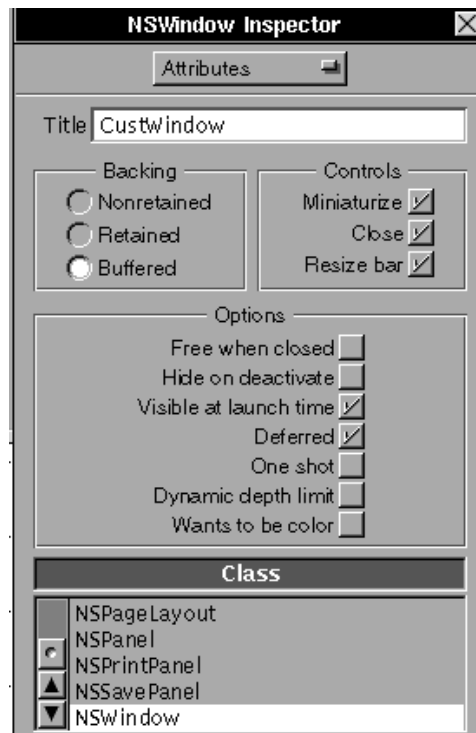


Figure 3-15 The Inspector Panel

You can control whether the palette window and the Inspector panel appear when Interface Builder starts by checking the appropriate boxes in the Preferences panel.

You use the Inspector panel to edit the properties of both `NSView` and non-`NSView` objects in your application. You can use the panel to do the following:

- Display information about an object
- Set an object's attributes
- Connect two objects
- Display a help topic attached to an object or attach a help topic to an object

The Inspector panel is displayed when you choose the Inspector command in Interface Builder's Tools menu.

The panel has many personalities. Its contents are determined by Interface Builder's selection: If an `NSButton` object is selected, the Inspector panel displays the `NSButton` Inspector; if an `NSWindow` is selected, the panel displays the `NSWindow` inspector. (The Inspector panel's title announces the class of the selected object.) In addition, the panel itself has four displays—Attributes, Connections, Size, and Help—which are accessible through the pop-up list at the top of the panel. These four displays are discussed in “Setting Object Attributes” on page 3-63, “Sizing Windows and Panels” on page 3-36, “Positioning and Sizing Precisely” on page 3-40, “Automatically Resizing Objects” on page 3-98, “Making and Managing Connections” on page 3-109, and .

Creating a Nib File

To create a new nib file, do one of the following:

- Choose one of the panels in the New Modules submenu of the Document menu.

Or

- Choose New Empty from the New Modules submenu, then drag a window or panel from the Windows palette.

Or

- Choose New Application from the Document menu.

Nib files are created for you automatically when you create applications in Project Builder. But sometimes you need to create nib files directly in Interface Builder, typically when you want to add additional windows and panels to your application.

To create a new panel (and the nib file that contains it) choose the desired panel type from the New Module submenu of the Document menu. For example, if you choose New Info Panel, you get the template panel shown in Figure 3-16.

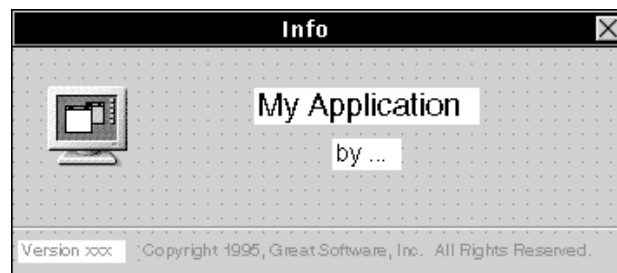


Figure 3-16 New Info Panel

Most commands of the New Module submenu create new nib files that contain a special kind of ready-made panel; your application can later load these nib files when it needs them. The New Empty command just creates an empty nib file; you must create the windows and panels for it by dragging these objects from the Windows palette. The New Application command in the Document menu can create your application's main nib file (a nib file with the owner of Application) if that has not already been done for you in Project Builder.

Note – You can have auxiliary nib files, such as an Info panel, that you load into your program only when you need to.

Saving the Nib File

An UNTITLED nib file window is displayed for each newly-created nib file. After you make changes to an interface, remember to save the nib file. Choose Save from the Document menu and specify a path and file name in the Save Panel as shown in Figure 3-17. Interface Builder may ask if you want to insert the file into your project; you usually confirm by clicking on Yes.

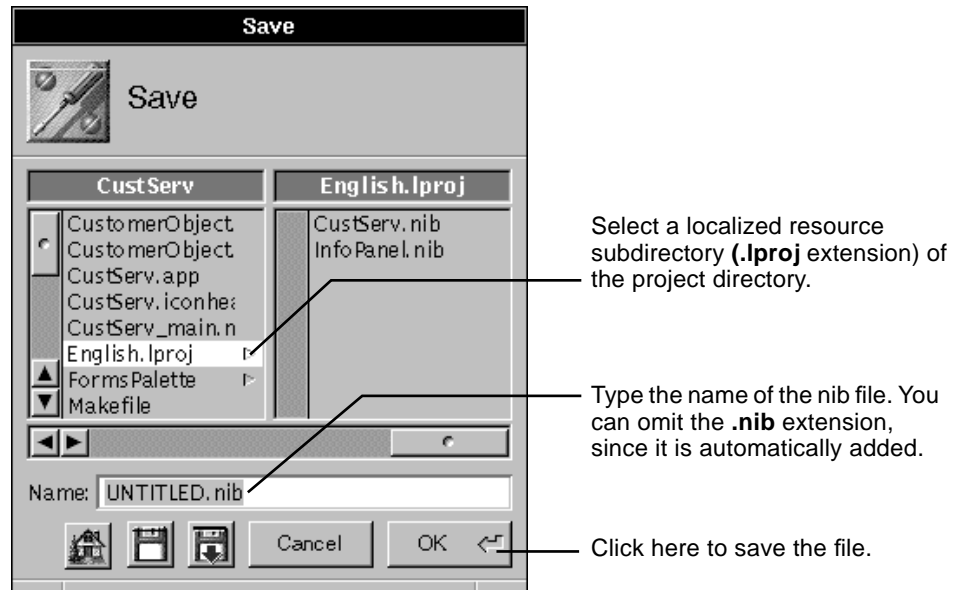


Figure 3-17 Saving a Nib File

What Is in a Nib File

When you save an interface in Interface Builder, it is archived to a nib file. Every application has a main nib file, which contains the main menu and often a window and other objects. A nib file (actually a file package) has the extension `.nib`. This nib file contains the following:

- Archived objects, in their hierarchy
- Sound and image data
- Information on custom classes
- Connection information

Archived Objects

The nib file stores encoded information on kit objects such as those shown in Figure 3-18, including their size, location, and position in the object hierarchy (for `NSView` objects, determined by `Superview/subview` relationship). At the

top of the hierarchy of archived objects is the File's Owner object, a proxy object that points to the actual object that owns the nib file. (For a description of File's Owner, see "The Nib File's Owner" on page 3-18.)



Figure 3-18 Archived Objects

Sounds and Images

Any sound or image files (TIFF or EPS) that you drag and drop over the nib file window are stored in the nib file and represented by the icons shown in Figure 3-19. The Sounds and Images displays of the nib file window are described in "Sounds Display" on page 3-16 and "Images Display" on page 3-15.



Figure 3-19 Sounds and Images

Class References

Interface Builder can store the details of kit objects and objects that you palettize (static palettes), but it does not know how to archive instances of your custom classes since it does not have access to the code. For these classes, Interface Builder stores a proxy object to which it attaches class information, as shown in the example in Figure 3-20 on page 3-26.

```
MyClass = {  
    ACTIONS = {  
        dothis;  
    };  
    OUTLETS = {  
        textField;  
    };  
    SUPERCLASS =  
        NSObject;
```

Figure 3-20 Custom Class Information

Connection Information

A nib file also contains information (such as that represented in Figure 3-21) about how objects within the object hierarchy are interconnected. Connector objects special to Interface Builder store this information. When you save the document, connector objects are archived in the nib file along with the objects they interconnect.

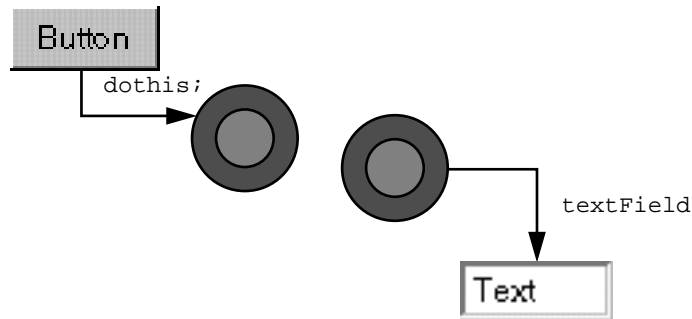


Figure 3-21 Connection Information

When You Load a Nib File

The following things take place when you load a nib file with the `loadNibSection:owner:withNames:` method:

- The run-time system unarchives the objects from the object hierarchy, allocating memory for each object and sending it a `read:` message. After its unarchived, an object receives `awake` and `finishUnarchiving` messages.
- It unarchives each proxy object and queries it to determine the identity of the class that the proxy represents. Then it creates an instance of this custom class (`alloc` and `init`) and frees the proxy.
- The system unarchives the connector objects and allows them to reestablish connections, including connections to File's Owner.
- As the final step, the run-time system sends `awakeFromNib` to all objects that were derived from information in the nib file, signalling that the loading process is complete.

Using the Palettes

Palettes store ready-made objects that you can add to your interface. Drag the object from the palette to add it to your interface.

When the grid in your interface is turned on and you add or move an object, the object snaps to the grid. Similarly, when you add a new object to the window or resize an object, the dimensions of the object snap to the grid.

The Palette window displays the palettes available to you. The window usually is displayed in the upper right corner when you start Interface Builder (see Figure 3-1). Choose Palettes from the Palettes submenu to display the Palette window if it is not visible. The Tools menu contains the Palettes submenu.

Each palette is represented in the window by an icon. If more than four palettes are loaded, a horizontal scroll bar gives access to those palette icons that are not visible. Click on an icon to display that palette.

The palettes for the Application Kit—the Menu palette, the Views palette, the TextViews palette, and the Windows palette—are loaded by default. These palettes provide windows, panels, browsers, scroll views, buttons, text fields, and a number of other interface objects.

You can create your own palettes of objects. Custom palettes that you create fall into two categories: static palettes, which are created as separate projects, and dynamic palettes, which you create while you create your interface. For more information on creating palettes, see “Adding Custom Palettes, Inspectors, and Editors” on page 3-169.

The Menu Palette

Add a menu cell to your interface by dragging it from the Menu palette, shown in Figure 3-22, into your interface's menu.

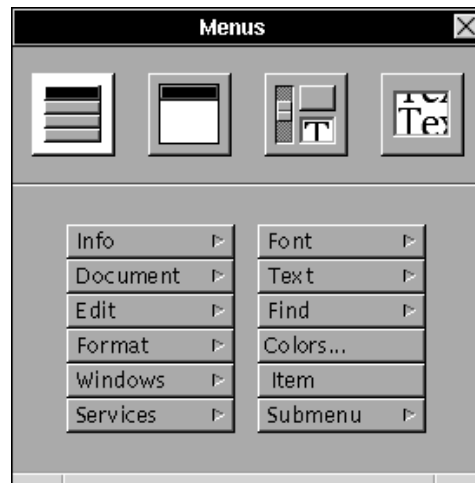


Figure 3-22 The Menu Palette

All of the menu cells on this palette create standard OpenStep menus except for the Item and Submenu cells.

See *OpenStep Programming Reference* for the types of messages a menu cell can receive.

The Views Palette

Create an `NSView` object by dragging it from the Views palette into a window or panel in your interface. The Views palette, shown in Figure 3-23, contains objects created from subclasses of the `NSView` classes.



Figure 3-23 The Views Palette

You can create an `NSMatrix` from an `NSTextField`, `NSForm`, `NSSlider`, or `NSButton` object by holding down the `Alt` key and dragging one of the resize handles of the selected object until several copies are made. You can also add and delete rows and columns in the same manner.

The individual objects within an `NSMatrix` are `NSTextFieldCells`, `NSFormCells`, `NSSliderCells`, or `NSButtonCells`. You can set attributes for the `NSMatrix` object that apply to all of the cells, or for each cell individually.

Copying an `NSView` object copies all of the `NSView` objects within that object (its subviews) as well.

`NSView` objects that display text appear on the TextViews palette (see “The TextViews Palette” on page 3-30).

Use the `NSCustomView` object to create an object from a subclass of `NSView` that does not appear on the Views palette or the TextViews palette.

The TextViews Palette

Add a text viewer to your interface by dragging it from the TextViews palette, shown in Figure 3-24, onto a window or panel in your interface.

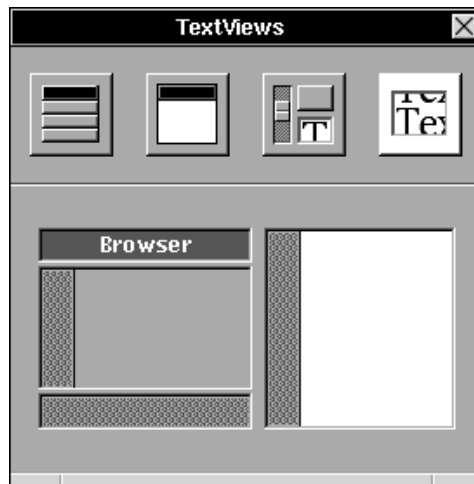


Figure 3-24 The TextViews Palette

The Windows Palette

Add a window or a panel to your interface by dragging it from the Windows palette, shown in Figure 3-25, to anywhere in the workspace.

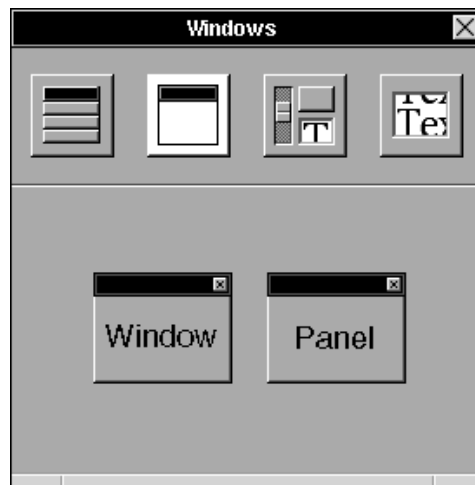


Figure 3-25 The Windows Palette

To use a subclass of `NSWindow` other than `NSPanel`, drag an `NSWindow` object into the workspace, select it, and change its class using the Inspector panel.

See *OpenStep Programming Reference* for the types of messages an `NSWindow` or `NSPanel` object can receive.

Adding an Object from a Palette to Your Interface

To add an object from a palette to your application's interface, do the following, as shown in Figure 3-26:

1. Choose the palette you want.
2. Drag an object from the palette to the appropriate "surface."
3. Release the mouse button.

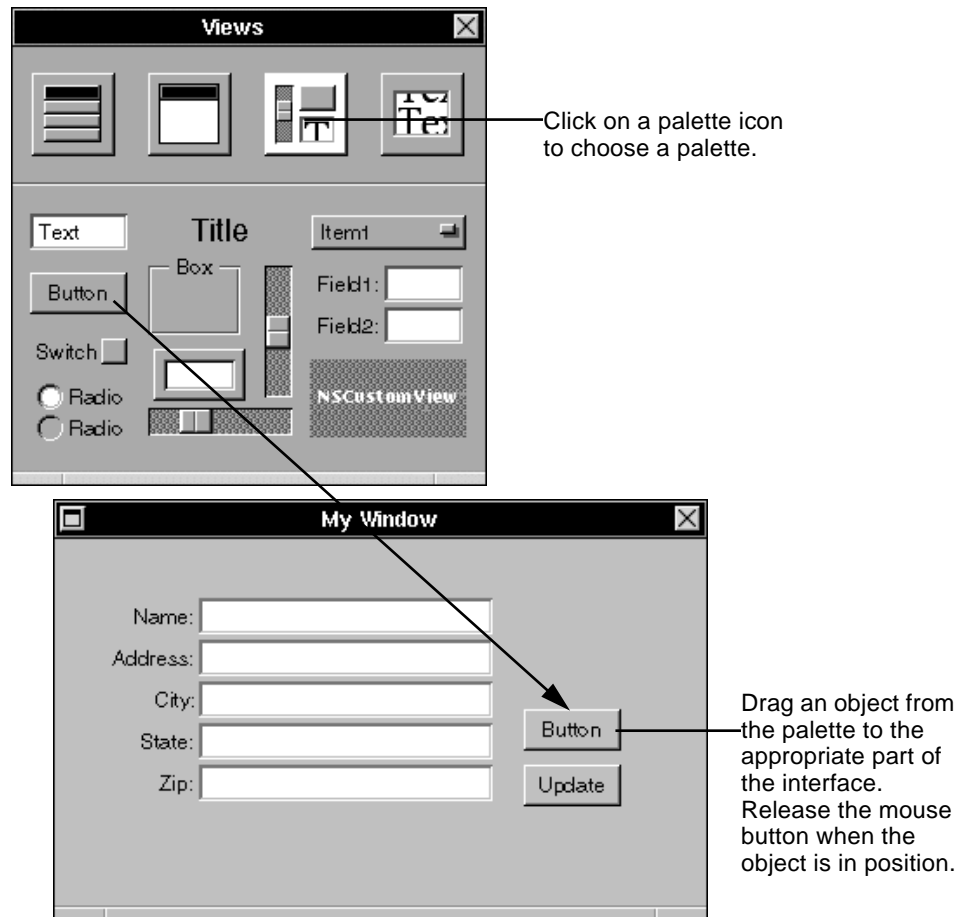


Figure 3-26 Dragging an Object from a Palette to the Application Interface

Note – Where you "drop" a window or panel is important, since that sets its initial position on the screen, the location where it is displayed when the application starts or when its nib file is loaded.

"Where Palette Objects Go" on page 3-34 illustrates the proper "surfaces" for interface objects.

Placing Interface Objects

To move an interface object within a window or panel in your application's interface, do the following, as shown in Figure 3-27:

1. **Select the object you want to move.**
2. **Drag the object to the new location in the window or panel.**

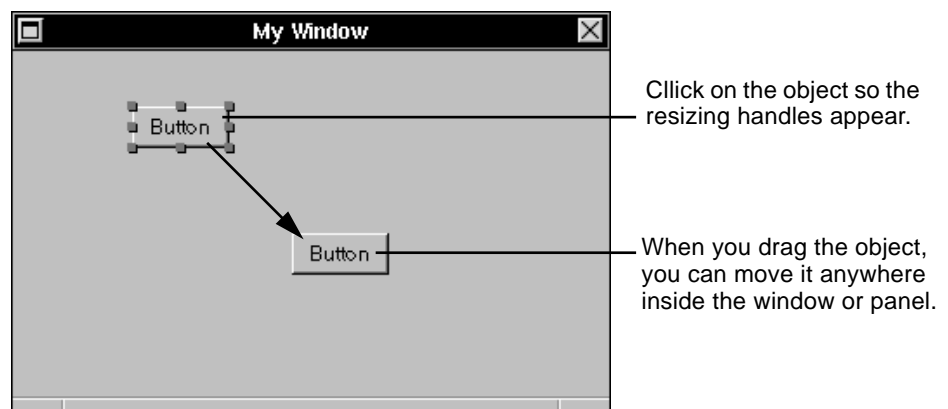


Figure 3-27 Placing an Interface Object

To move an object around the "surface" of a window or panel, select the object and drag it with the mouse. The currently selected object has resizing handles around its perimeter.

When you move an object, make sure that the mouse pointer is inside the object and not on a resize handle.

For greater precision, select an object and press the arrow keys; this moves the object an incremental distance in the required direction. If the alignment grid is off, this distance is one pixel; if it is on, it is the distance of the grid.

You can adjust the size and location of objects precisely by specifying their origins, width, and height in the Size display of the object's Inspector. See "Positioning and Sizing Precisely" on page 3-40 for details.

Selecting Multiple Objects

You can select multiple objects and then move, copy, or do other things with them as a group. There are two ways to select more than one object:

- Hold down the Shift key while you click on objects in succession.
- Click in an empty area, then draw a "rubberbanding" rectangle around all objects you want selected.

After making the selection, press (do not momentarily click) the mouse pointer on one of the objects and drag the group to the new location. (Or do another suitable operation, such as copy and paste.)

To deselect an object in a grouped selection, hold down the Shift key and click on that object.

You cannot do sizing operations on multiple selected objects.

To select all objects in a window or panel, first select the window or panel, and then choose the Select All command from the Edit menu. You can select all items in the Instances or Classes display by choosing Select All from the Edit menu. The command-key equivalent for Select All is Command-a.

Where Palette Objects Go

You can put windows and panels anywhere in the work space as shown in Figure 3-28. Nothing contains them except the screen.

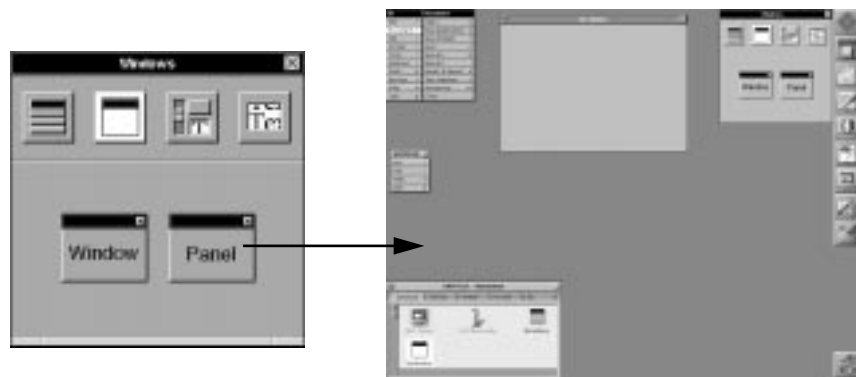


Figure 3-28 Putting a Panel in the Workspace

You put items from the Views and TextViews palettes— buttons, labels, pop-up menus, fields, boxes, text fields, scroll views, browsers, custom `NSViews`— anywhere within the bounds of a window or panel as shown in Figure 3-29.

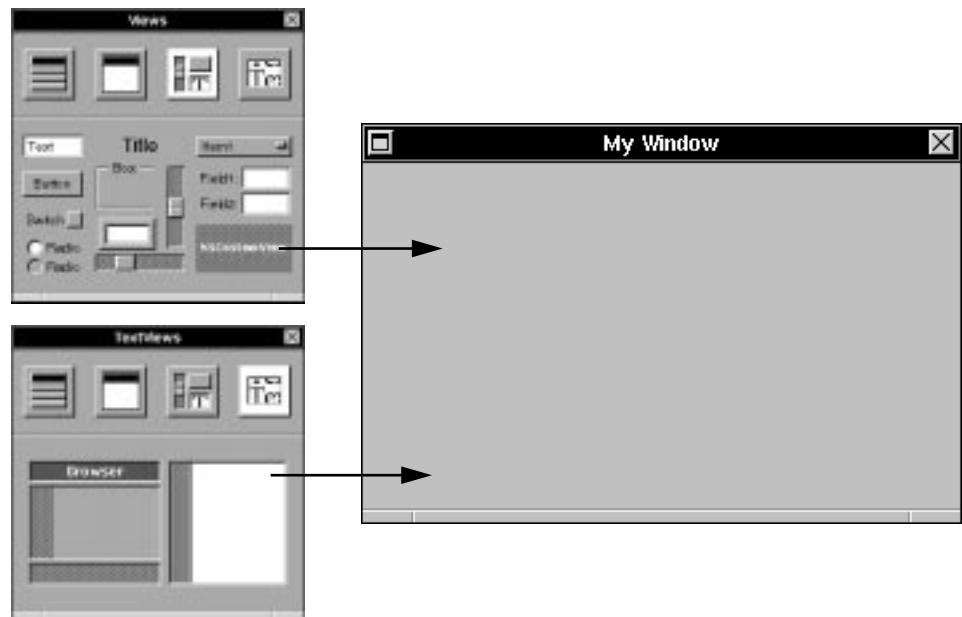


Figure 3-29 Putting `NSViews` and `NSTextView`s in a Window

You drag a menu cell from the Menu palette and drop it in the application's menu as shown in Figure 3-30. When you release the mouse button, Interface Builder inserts the cell between the two menu commands underneath it.

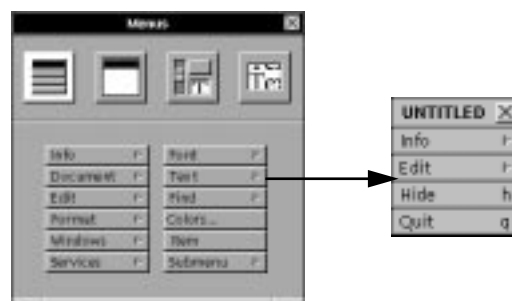


Figure 3-30 Putting a Menu Cell in the Application's Menu

Sizing Windows and Panels

After you drag a window or panel from the Windows palette and drop it on the screen, you will probably want to resize the object to a suitable dimension. To resize a window or panel, do one of the following:

- Drag the resize bar in the direction you want the window to grow.

Or

- Bring up the Inspector panel and enter the dimensions in the Size display.

To resize a window, drag the resize bar in the required direction as shown in Figure 3-31.

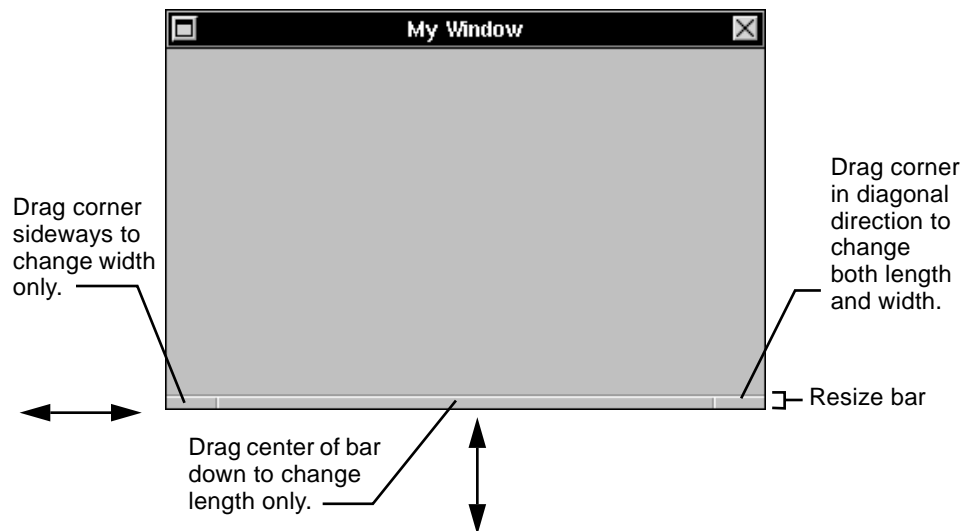


Figure 3-31 Sizing a Window with the Resize Bar

Many panels and some windows are not set for resizing and therefore do not have a resize bar visible. To make this bar temporarily visible for resizing in Interface Builder, check the miniaturize switch button in the Controls section of the Attributes inspector for the window.

You can also resize windows and panels with greater precision by entering the exact dimensions in the Size display of the Inspector panel as shown in Figure 3-32. To bring up the Size Inspector for a window, select the window by clicking on its title bar, then choose Inspector from the Tools menu (or press Command-3).

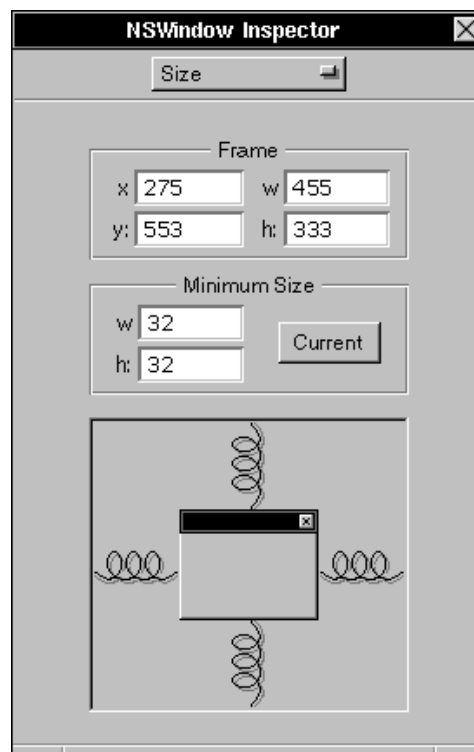


Figure 3-32 Sizing a Window with the Size Display of the Inspector Panel

You can also use the Inspector panel to size `NSView` objects with numerical exactness. See “Positioning and Sizing Precisely” on page 3-40 for further information.

Also see “The Coordinate System in Interface Builder” on page 3-49 for some conceptual background.

Initializing Text

Many of the palette objects include text as a component. Buttons of all sorts usually have titles, boxes usually name the elements they group, and so on. Interface Builder initially sets the text in most of these objects to the name of the object itself (such as "Button" or "Text"). After you drag the palette object onto a window or panel, you will probably want to delete these text strings or rename them to something meaningful. This text is what is initially displayed when your application loads the nib file; your application can later change the text.

To change the text in an interface object, do the following, as shown in Figure 3-33:

1. **Select the object.**
2. **Double-click on the text inside the interface object.**
3. **Edit the text.**
4. **Deselect the text by clicking outside of it.**



Double-click on the text to select it.



Type the new text. When finished, click outside the object to set the text.

Figure 3-33 Editing the Text on an NSButton (Switch) Object

Once text is selected, you can move the pointer among the characters by pressing the left and right arrow keys; you can delete characters by pressing the Delete key. Text fields are initialized to "Text" (which you will almost always want to delete). To delete this, double-click on it and press the Delete key.

Matrices—compound objects, such as radio buttons and form fields—need a slightly different procedure for selecting text for initialization: You must double-click on the embedded text item twice, the first time to select the embedded object, and the second time to select the text inside the object.



In `NSMatrix` objects, double-click on the text to select the embedded object.



Double-click again to select the text.

Figure 3-34 Editing the Text on an `NSMatrix` Object

“Creating Matrices of Objects” on page 3-58 describes how to create these compound objects. Also see “Compound Objects” on page 3-91 for a conceptual summary of `NSMatrix` objects and other compound objects.

Sizing Interface Objects

Interface objects in Interface Builder scale to any practical dimension. You can, for instance, increase the size of a button so it fills a window. Most interface objects, however, do not scale below a certain minimum size of usefulness.

To size an interface object, do the following:

- 1. Select an object.**
- 2. Drag a resize handle in the desired direction.**

To size an object you must first select it. A selected object has resize handles—small, gray rectangles—around its perimeter. Drag one of these handles in the direction you want the object to increase (or decrease) in size.

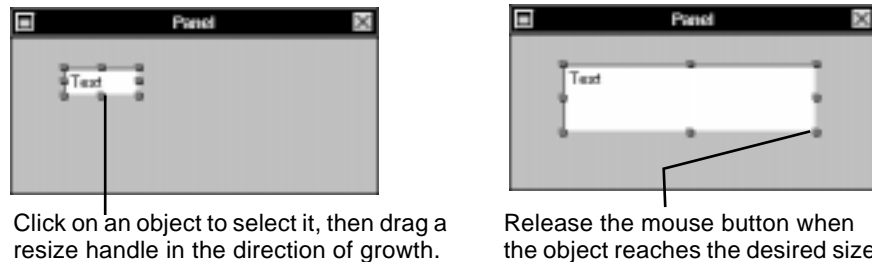


Figure 3-35 Sizing an Interface Object with its Resize Handles

To affect just one dimension of the object, drag a top, bottom or side handle. To adjust both dimensions simultaneously, drag one of the corner handles. To size both dimensions proportionally, hold down the Shift key while you drag a corner resize handle.

You can adjust the size and location of objects precisely by specifying their origins, width, and height in the Size display of the object's Inspector panel.

See "Positioning and Sizing Precisely" below for details.

Positioning and Sizing Precisely

You can move and resize objects in your interface with numerical exactness using the Inspectors for those objects. You will occasionally find need for such exactness, such as when you want to size a custom view to the same dimensions as the image that it will display. More frequently you will use this method to align objects or make sure they are the same size.

To position or size an object precisely, do the following:

- 1. Select an object.**
- 2. Choose the Size Inspector for that object.**
- 3. Modify the object's origin point or its dimensions.**

To size and move objects with precision, select a reference object and then choose the Inspector option from the Tools menu. Choose Size from the pop-up menu at the top of the Inspector panel. (You can also bring up the Size display by pressing Command-3.) Note the position and dimensions of the reference object.

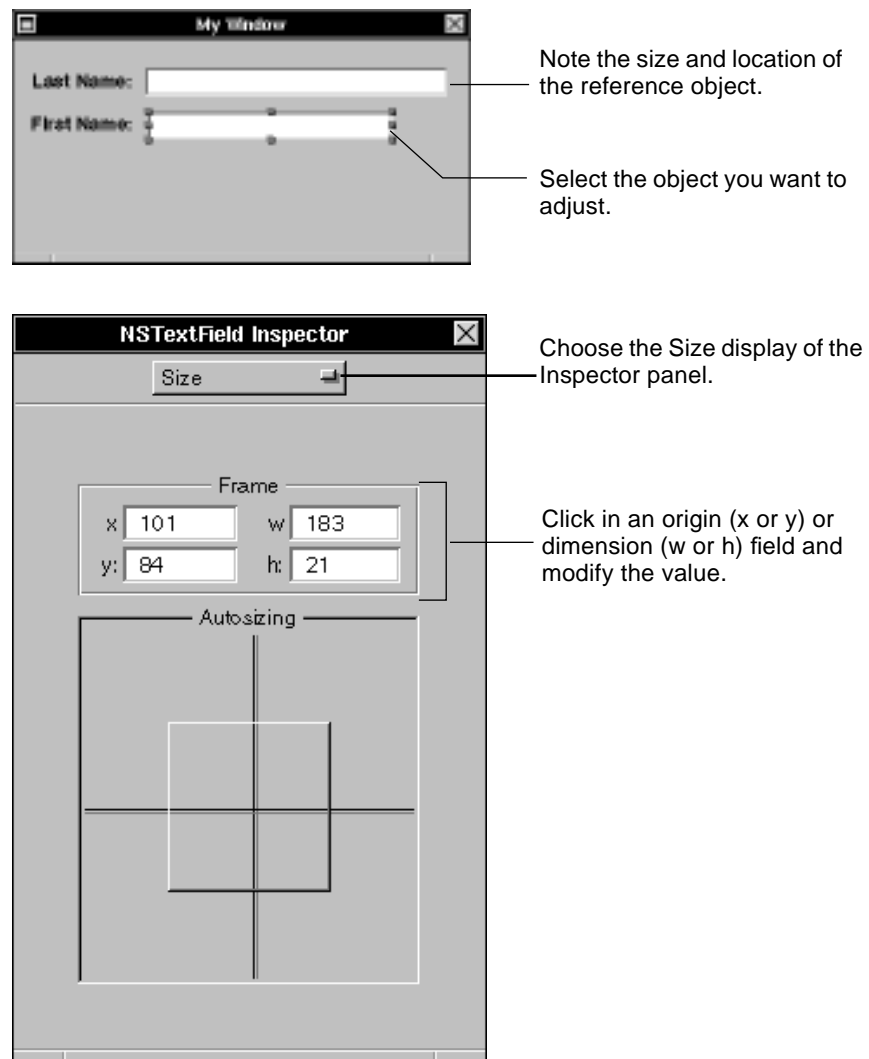


Figure 3-36 Sizing an Interface Object with the Inspector Panel

When you press Return in an origin or dimension field, the object moves to the new position or expands or contracts to the new size.

Note - You can also move selected objects incrementally—and precisely—by pressing the arrow key that points in the required direction. Each incremental "nudge" moves the object the distance of the grid or, if the grid is turned off, one pixel.

Duplicating Objects

You can duplicate an object in your application's interface by doing the following:

- 1. Select an object.**
- 2. Copy the object to the pasteboard.**
- 3. Paste the object back to the interface.**
- 4. Position the new object.**

To duplicate an object, select it (see Figure 3-37) and then copy and paste it just as you would with geometric shapes in a drawing application. The copied object has the dimensions and most other attributes of the original object.

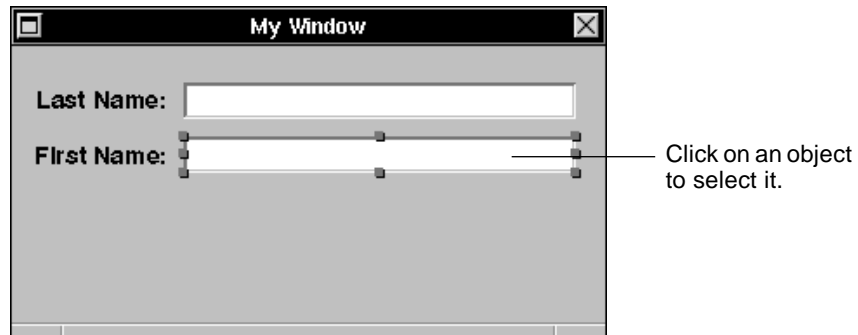


Figure 3-37 Selecting an Object to Duplicate

Choose Copy from the Edit menu, then choose Paste from the Edit menu.

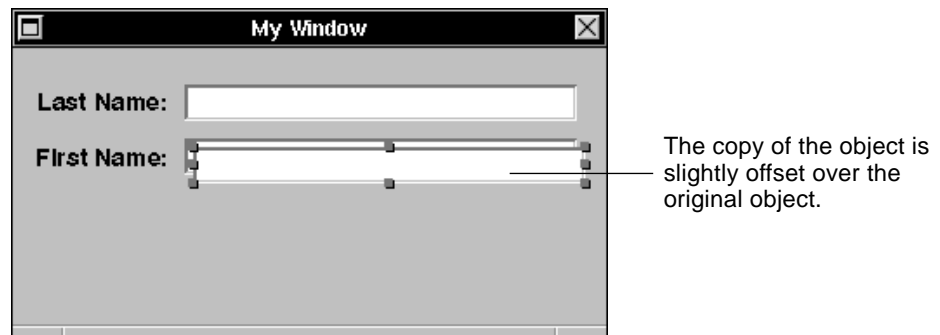


Figure 3-38 The New Object After Duplication

The new object is offset slightly from the original one to help you select it, as shown in Figure 3-38. Move the new object to its new location.

In addition to the objects that appear on the interface, you can copy your custom non-UI objects—represented as cubes in the icon mode of the nib file window—as well as your windows and panels. Just click to select them and then copy and paste them.

Note – Instead of choosing Copy and Paste from the menu, you can press Command-c (Copy) and Command-v (Paste).

You can also duplicate groups of selected objects by copying them and then pasting them. See “Selecting Multiple Objects” on page 3-34 for details on making multiple selections of objects.

Moving Objects to Other Windows

You can move an object from one window or panel in your application's interface to another by doing the following:

- 1. Select one or more objects.**
- 2. Alt-drag the objects to the other window or panel.**

To move objects from one window or panel to another, drag them between windows while holding down the Alt key, as shown in Figure 3-39.

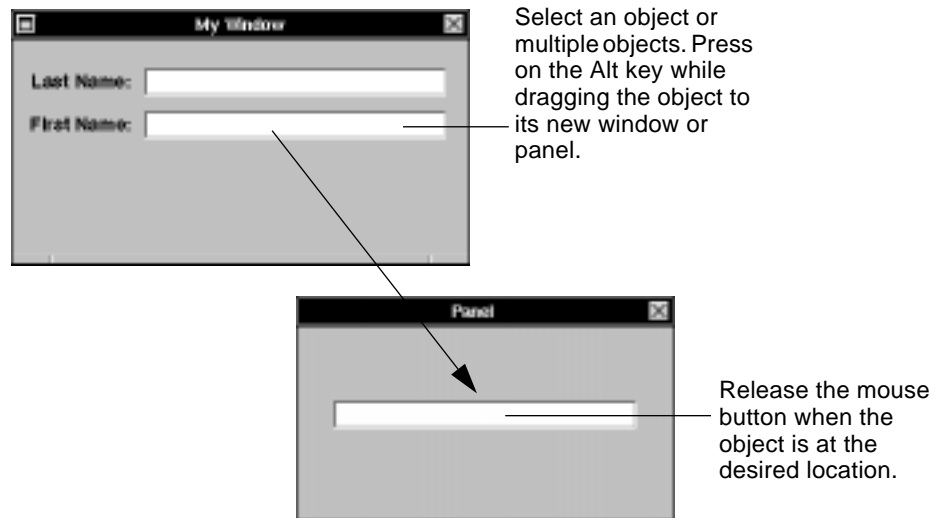


Figure 3-39 Moving an Object to Another Window

If you want to copy rather than move the selected objects (in other words, the original objects remain in the original location), you have two alternatives:

- Copy the objects using the Copy command; click in the other window or panel to activate it, and use the Paste command to copy the objects from the pasteboard.
- Copy and paste the objects in one window, then Alt-drag the duplicated objects to their new window or panel.

Copying Objects to Other Interfaces

With the same Alt-drag technique, you can copy objects between different nib files. Simply select a group of objects in one nib file and, while pressing the Alt key, drag and drop those objects on the appropriate "surface" of the other nib file.

Both nib files must be open when you initiate the copy operation. You can copy entire windows or panels as well as custom, non-UI objects between interfaces.

The surface onto which you drop objects must be compatible in the following ways:

- Non-UI objects must be dropped over the Instances display of the nib file window.
- View objects are dropped over a window or panel or over the Instances display.
- Windows and panels can be dropped anywhere on the screen.

The basic technique of Alt-drag also copies the connections among selected objects.

Arranging Objects

When you compose your interface, you usually want to arrange the objects in that interface in some appealingly regular way. You want buttons, for instance, to be aligned on the same invisible horizontal or vertical line. Or you want the distance between text fields in a form application to be exactly the same. Interface Builder gives you a set of tools for arranging objects.

To arrange objects in your application's interface, do the following:

- 1. Set the characteristics of the grid in the Alignment panel.**
- 2. Turn on the grid.**
- 3. Align objects with the grid.**

Every window or panel has a grid associated with it. You may turn this grid off and on. When it is on and you move an object, an edge of that object "snaps," like a nail to a magnet, to the adjacent intersecting lines of the grid.

Using the Alignment Panel

You set the dimensions of this grid and the edges of alignment in Interface Builder's Alignment panel, shown in Figure 3-40. To bring up this panel, choose the Alignment command in the Align menu (the Align menu is a submenu of the Format menu).

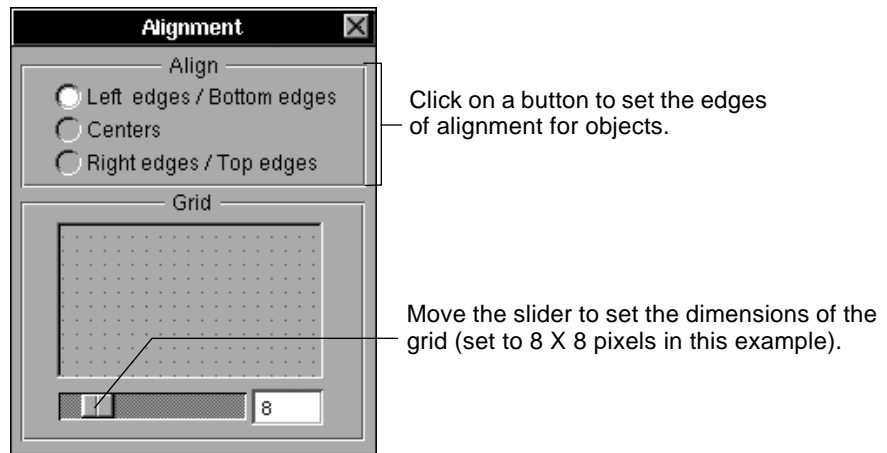


Figure 3-40 Using the Alignment Panel

The buttons in the Align section of the Alignment panel determine what point or edge of interface objects snaps to the grid (see Figure 3-41).

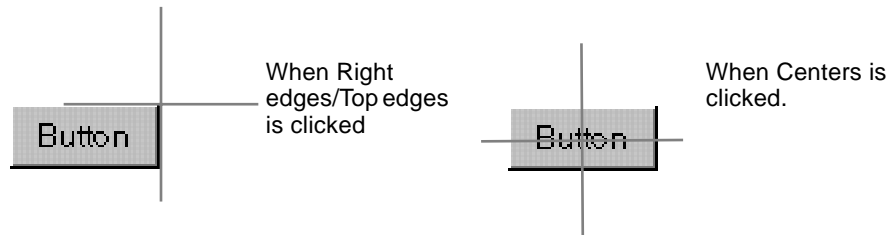


Figure 3-41 Using the Radio Buttons in the Alignment Panel

Once you have your grid set up, make sure the grid is turned on: Choose **Format** ▸ **Align** ▸ **Set Grid On**. If you also want the grid visible, choose **Show Grid** from the same menu.

Now align the objects, either individually or as a group, using the grid, as shown in Figure 3-42.

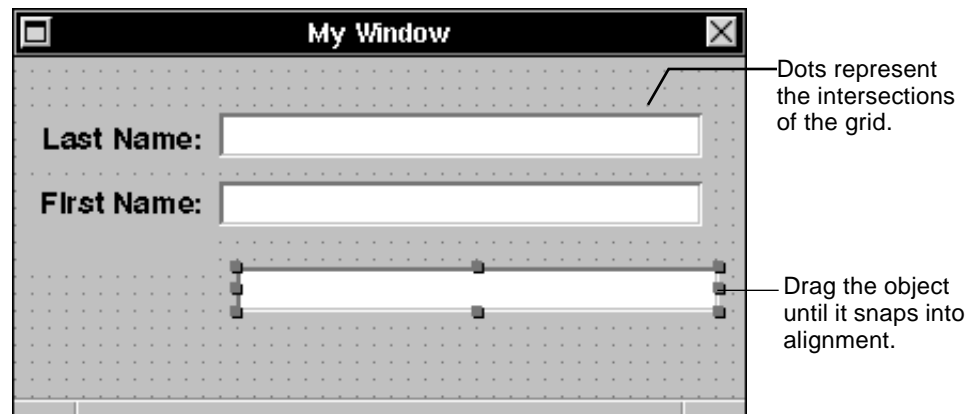


Figure 3-42 Aligning Objects Using the Grid

There are other ways to align objects that do not require using the mouse. With the grid turned off, you can drag view objects from a palette and visually align them as precisely as possible. Then set the grid spacing, turn the grid on, and choose the Align To Grid command.

Once the grid is set and on, align the objects, either individually or as a group, as shown in Figure 3-43.

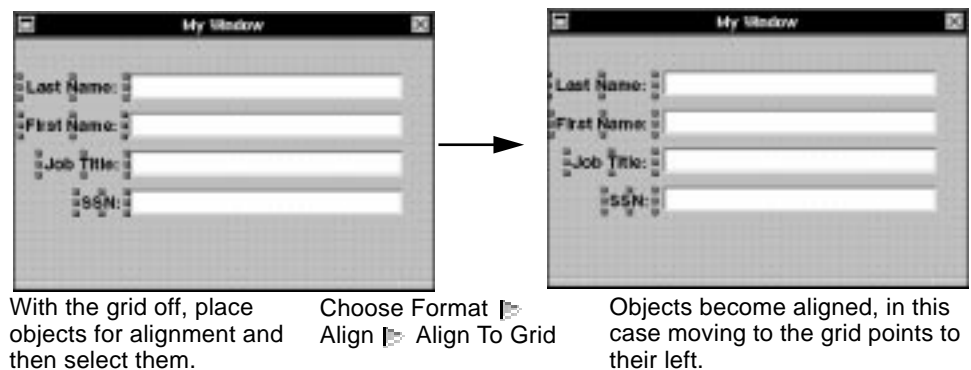


Figure 3-43 Aligning Objects to the Grid

With the Align To Grid command, the direction of alignment is toward the origin point of the window or panel (in other words, toward the lower-left corner). You should be aware of this when placing objects for later alignment.

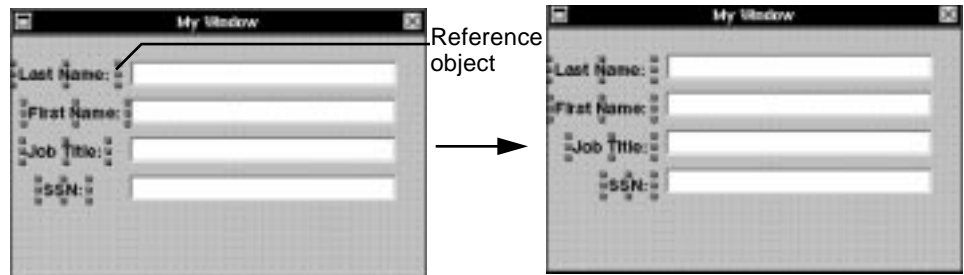
Note – You can align selected objects to a grid, singly or as a group, by pressing the arrow keys in the direction of alignment. When the grid is turned on, the unit of increment changes from one pixel to whatever the grid spacing is.

Making Columns and Rows of Objects

It is more efficient to align groups of objects than to align single objects successively. With the Make Column and Make Row commands, Interface Builder aligns groups of selected objects to a reference object. You designate the reference object by the way you select multiple objects:

- If you press the Shift key while clicking on objects in succession, the first object clicked on is the reference object (see Figure 3-44).
- If you draw a selection rectangle around a group of objects, and so select objects simultaneously, the topmost object in the selection (usually the most recently added object) is the reference object.

For most purposes, Shift-clicking on objects is the preferred method because it permits more control.



Click on the reference object first, then Shift-click on the remaining objects to select them.

Choose Format 
 Align  Make Column

The objects become vertically aligned to the reference object.

Figure 3-44 Making a Column of Objects

Removing Objects

To delete an object or objects from your application's interface, do the following:

1. **Select one or more objects.**
2. **Choose Cut from the Edit menu.**

To delete objects from an interface, select the objects and choose the Cut command, as shown in Figure 3-45.

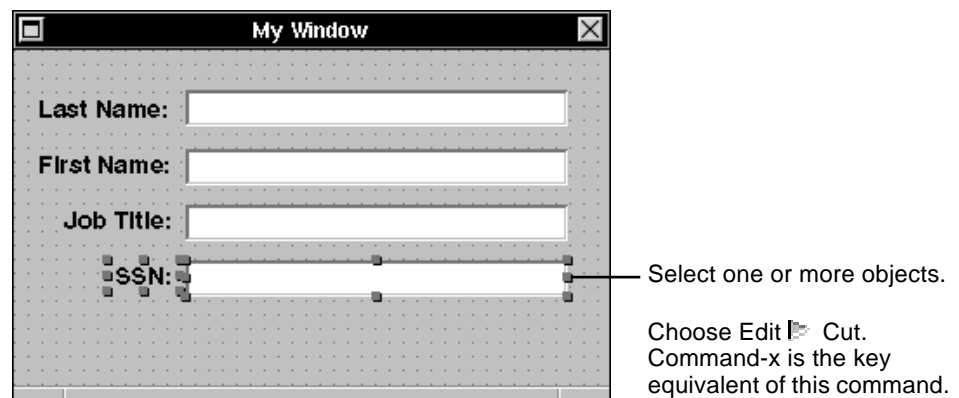


Figure 3-45 Deleting a Object from the Interface

You can also delete an object by pressing the Delete key, but the differences between the Delete key and the Cut command are significant. The Cut command saves the selected objects to the pasteboard, so you can retrieve the objects with the Paste command (Command-v). The Delete key permanently deletes the selected items.

The Coordinate System in Interface Builder

The Size display of an object's Inspector panel shows that object's precise location and dimensions. The x and y fields hold the origin point (horizontal and vertical) for the object within the drawing context of its enclosing window or panel. The w and h fields hold the width and height. All values are in pixels.

Within a window or panel, the lower left corner is origin 0,0. This is the point of reference for objects within that window or panel.

Therefore, when you move or size objects downward or to the left, the values in the Size display are decremented.

Figure 3-46 illustrates Interface Builder's coordinate system. The point of reference for a window or panel (or origin 0,0) is the lower-left corner of the screen. This means that the same relationship applies: if you decrement its x value in the Size display, it moves to the left; if you decrement its y value, it moves toward the bottom of the screen; decrement its w or h values and it becomes smaller.

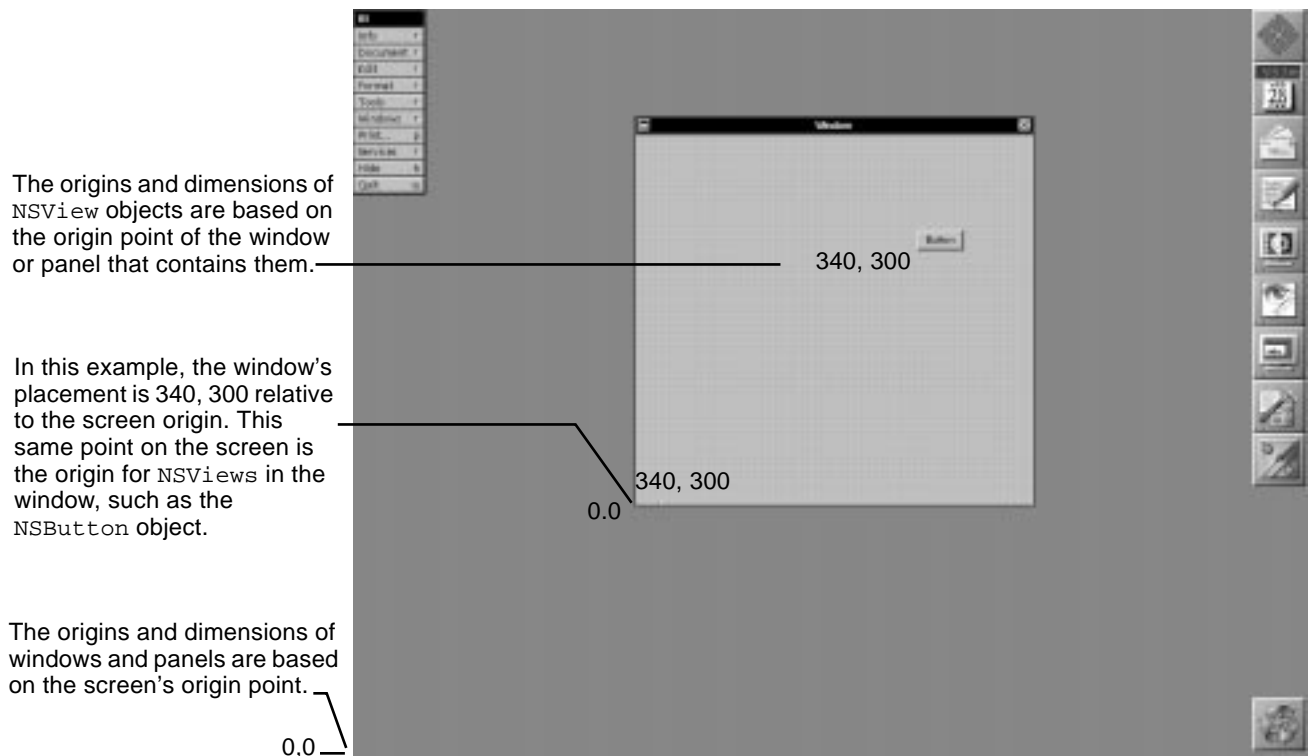


Figure 3-46 Interface Builder's Coordinate System

OpenStep's Basic UI Design Philosophy

Composing a user interface involves much more than techniques for placing, sizing, and arranging objects on a window. When you put your application's UI together in Interface Builder, keep in mind the following principles.

Make It Consistent

When all applications share the same basic interface, each application benefits. Consistency makes each application easier to learn, and so increases the likelihood of acceptance and use. Just as with so many natural "interfaces" in life, conventions count for a great deal. Although different applications are designed to accomplish different tasks, they all share, to some degree, a set of common operations such as selecting, editing, scrolling, and setting options. Reliable conventions are possible only when these operations are carried out the same way for all applications.

Make it Feel Natural

Try to make the screen a visual metaphor for the real world, so that the objects in it reflect the way the represented things actually behave. That is what an "intuitive" interface is—it behaves as we expect based on our experience with objects in the real world.

Modeled objects do not have to mimic every detail of their real counterparts, but they should behave in similar ways. For example, objects in the real world stay where we put them; they do not disappear and reappear again, unless someone causes them to do so. Users should immediately recognize the objects in your interface and should use them for the sorts of operations for which people typically use their real counterparts.

Put the User in Charge

Users should have the widest freedom of action. If an action makes sense, your application should allow it. In particular, avoid setting up arbitrary modes, periods during which only certain actions are permitted. On occasion, however, modes are a reasonable way of solving a problem, particularly in the following forms:

- attention panels

- modal tools
- "spring-loaded" mode (while mouse or key down)

But these modes should be freely chosen, provide an easy way out, be visually apparent, and keep the user in control.

At the same time, you should try to anticipate what users will do and ease their way, reducing the actions they must perform. Give them freedom, but still act on their behalf without waiting for their instructions. These helping actions should be simple and convenient, like, in the Open panel, preselecting a directory that is probably in the path of the final selection.

Focus on the Mouse

The mouse is the most appropriate instrument for a graphical interface. The keyboard is principally used for entering text, but the mouse is the instrument by which users manipulate the objects of your interface. Your user interface should support the following three paradigms of mouse action:

- Direct manipulation
- Targeted action
- Modal tool

See *User Interface Guidelines* for more on action paradigms and much more information important to the design of your user interface.

Making Interface Objects the Same Size

To lend a look of consistency to your interface, you often want to make similar objects the same size. Buttons across the bottom of an attention panel, for instance, should be the same exact size. Interface Builder gives you an easy way to do this, allowing you resize selected objects to a reference object. You designate the reference object differently, depending on your method of selection:

- If you press the Shift key while clicking objects in succession, the first object clicked is the reference object.
- If you draw a selection rectangle around a group of objects, selecting the objects simultaneously, the topmost object in the selection (usually the most recently added object) is the reference object.

To make one or more objects the same size as a reference object, do the following:

1. **Select the reference object.**
2. **Add to the same selection the objects that you want resized.**
3. **Choose the Same Size command.**

Making objects the same size involves identifying and selecting a reference object and selecting several other objects (as shown in Figure 3-47), and choosing a command.

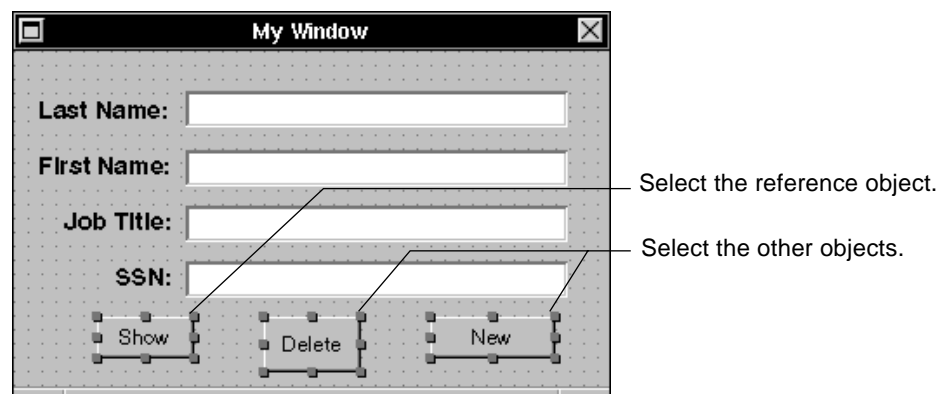


Figure 3-47 Selecting Several Objects and a Reference Object

Choose Same Size from the Size submenu (you can find the Size submenu on the Format menu). The objects become the same size as the reference object, as shown in Figure 3-48.

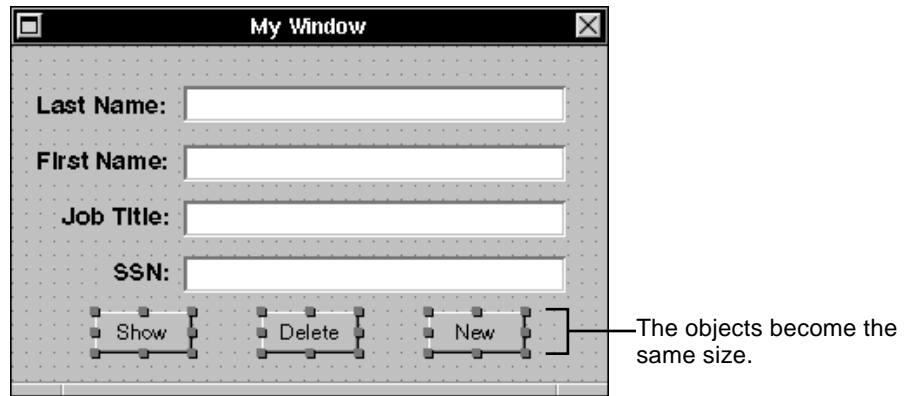


Figure 3-48 The Objects Become the Same Size as the Reference Object

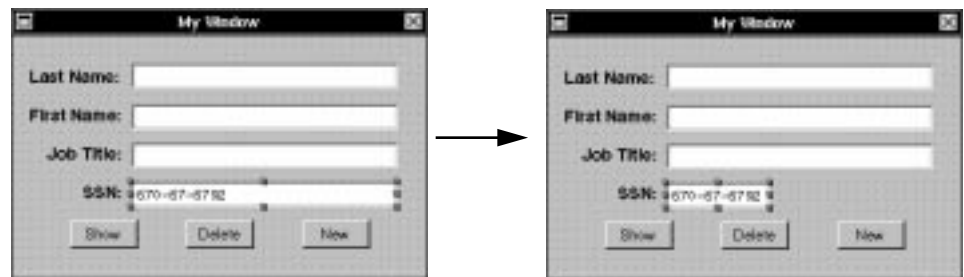
Note – In most situations, you should select multiple objects by Shift-clicking them because this method gives you more control (you do not always have to keep track of the topmost object as the reference object).

Shrinking Objects to their Minimum Size

To conserve screen real estate, or to enhance the appearance of your interface, you might want to have `NSView` objects just large enough for any text they contain. You can do this with the Size to Fit command.

To make an object or objects the size needed to accommodate their text contents, do the following (see Figure 3-49):

1. **Select one or more objects.**
2. **Choose Size to Fit.**



Select the object you want to shrink.

Choose Format ▸
Size ▸ Size To Fit

The object shrinks to enclose the text.

Figure 3-49 Sizing an NSView Object to Fit the Text It Contains

The Size to Fit command has no effect on matrices, custom views, and some other objects. If you delete the text from a button, text field, or other object that holds text, and then apply the Size to Fit command to it, that object shrinks to its minimum (and probably unusable) size.

Grouping Objects

You can group a set of objects in your application's interface by doing the following:

1. **Select the objects you want grouped.**
2. **Choose the Group command.**

When you group objects, Interface Builder draws a box around them. The box has a title (initially "Title"). You select, move, copy, and cut and paste the objects within the box as a group. Interface Builder gives you two ways to group objects.

In the first method, you select the objects of the group and choose a menu command (see Figure 3-50).

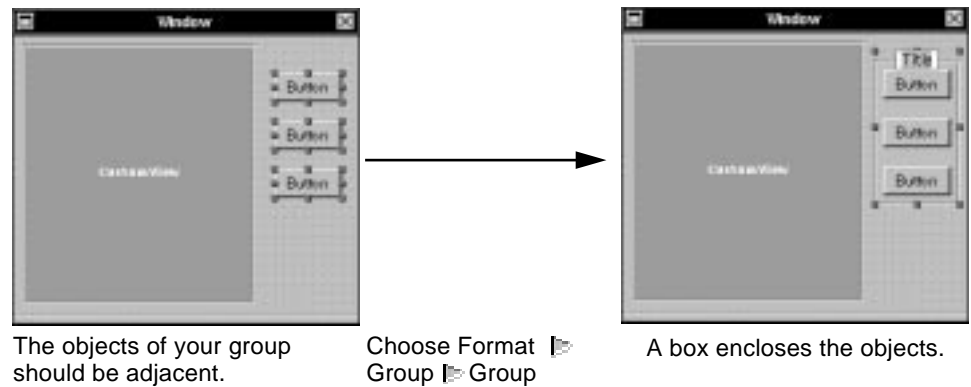


Figure 3-50 Selecting Objects and Using the Group Command

The default title of the box around the grouped objects is "Title." To change this, double-click the title to select it (as in the window on the right side of Figure 3-50). Then type the new name for the grouped objects.

To ungroup the objects, making each object individually selectable again, select the group and choose Ungroup from the Group submenu.

You can also use the NSBox object in the Views palette to group objects (see Figure 3-51).

1. First, drag a box onto a window or panel.
2. Then add its contents.

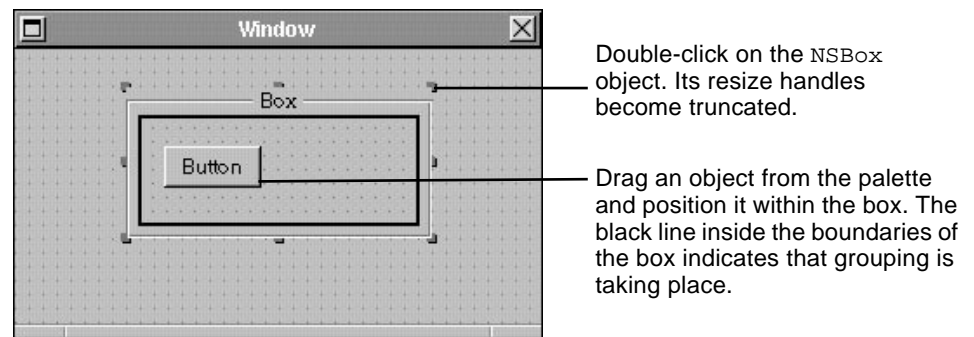


Figure 3-51 Using an NSBox Object to Group Objects

The default title of the `NSBox` object is "Box." To change this, double-click the title to select it. Then type the new name. You can change the position of the title, or eliminate the title altogether, using the `NSBox` Attributes inspector (see Figure 3-74).

The Group submenu has two other interesting commands. With the Group in ScrollView command, you can automatically bind an `NSText` object (or your own custom `NSView` object) to horizontal and vertical scrollbars. With the Group in SplitView, you can group two related views (often a custom view and a browser object) in a split view, which has a sizing bar between the views.

See the specifications of the `NSScrollView` and `NSSplitView` classes in *OpenStep Programming Reference*.

Layering Objects

Every object on a window or panel in Interface Builder is on its own layer. That is why when you move one object over another object, the first object moves in front of the second or moves behind it. The most recently added object is generally on the topmost layer. To change the layering order of an object, do the following:

- 1. Select an object.**
- 2. Choose Bring To Front or Send To Back.**

Occasionally, you need to alter the layering order to make an object visible or to have it appear behind other objects. To do this, apply the Bring To Front command or the Send To Back command (on the Format menu) to selected objects.

For example, assume you want two buttons to partly overlay a `NSScrollView` object, as shown in Figure 3-52.

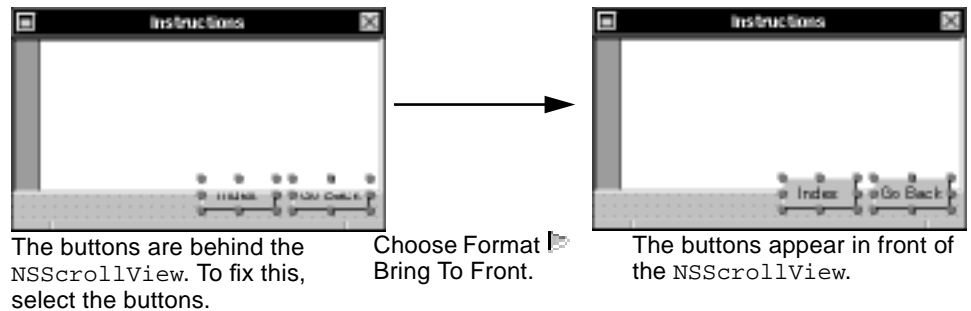


Figure 3-52 Layering Buttons in Front of an NSScrollView Object

Creating Matrices of Objects

You can easily transform certain objects in the standard Interface Builder palettes into matrices of those objects. A matrix (defined by class `NSMatrix`) imposes a regular size and intervening distance on a set of identical objects. Matrices afford an easy way to compose forms, arrays of buttons and sliders, and multiple-column browsers.

You can create a matrix of objects by doing the following:

1. **Drag a suitable object from the Views or TextViews palette.**
2. **Alt-drag a resize handle of the object.**

To create a matrix, drag one of these objects to a window or panel:

- text field
- button
- switch button
- radio button
- form field
- slider (vertical or horizontal)
- browser

Then size the object to the maximum dimension you anticipate for a cell in the matrix. Next create the matrix, as shown in Figure 3-53.

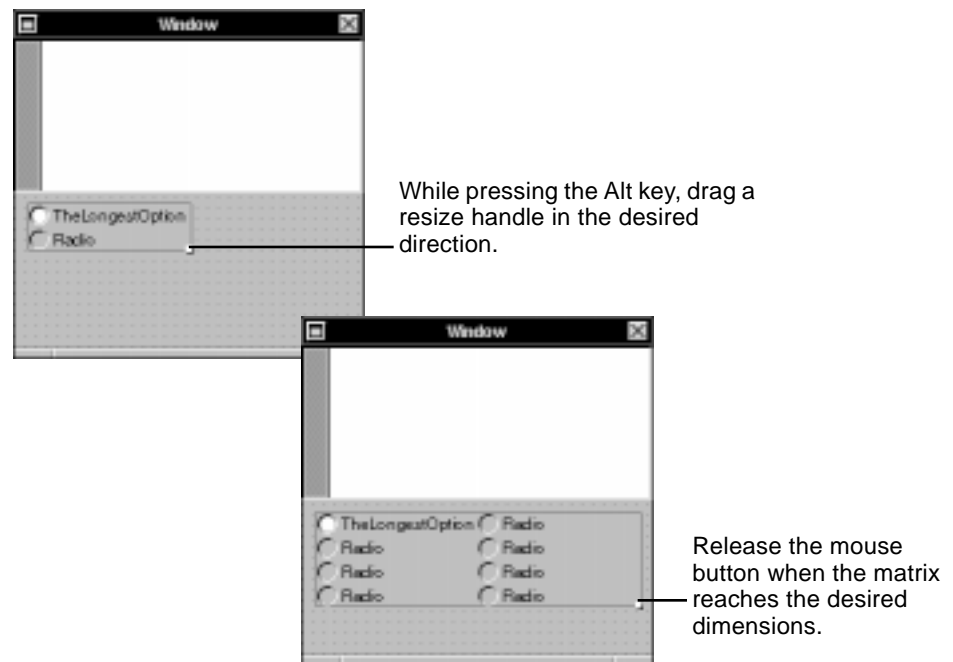


Figure 3-53 Creating a Matrix of Radio Buttons

Note – You can create a horizontal, vertical, or two-dimensional matrix of text fields, buttons, or sliders. A matrix of form fields can only be expanded vertically, adding fields at the bottom of the form.

Note – To make a browser with more than one column, drag a browser object from the TextFields palette onto your interface; then Alt-drag the right resize handle until the desired number of columns appear.

Creating Menus

Menus are just as important as windows and panels for an interface. Menu commands initiate most of the standard functions of an application, such as printing, opening files, or cutting and pasting text. That is why Interface Builder's Menu palette holds a number of ready-made submenus and menu cells.

To add a menu cell from the Menu palette to your application's menu, do the following, as shown in Figure 3-54:

1. Drag a menu cell from the Menu palette.
2. Drop it between two menu cells in your application's menu.

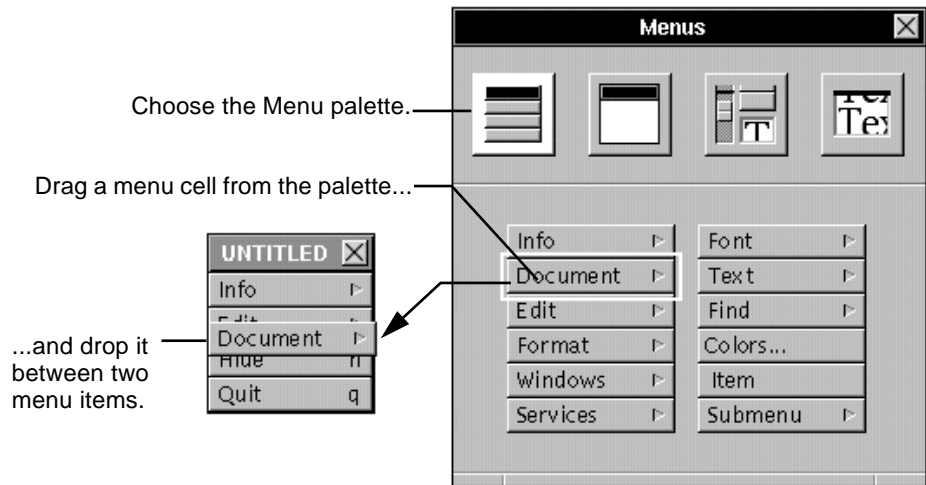


Figure 3-54 Adding a Menu Cell to the Application's Main Menu

Click on several menu cells in your application's main menu and note how some cells in the submenus are dimmed. Dimmed cells indicate that, as the default, the command is inactive until some condition occurs in your code that causes your application to activate the command.

Deleting a Menu Cell

You delete a menu cell just as you do with any other object in Interface Builder: select it, then choose the Cut command from the Edit menu (Command-x) or press the Delete key.

Changing Titles of Menu Cells

Also, as with other Interface Builder objects that display text, you can easily change the titles of menu cells by doing the following:

1. **Double-click on the text to select it.**
2. **Type the new title or edit the old one.**
3. **Click outside the cell to set the new title.**

Resequencing Menu Cells and Assigning Command Key Equivalents

You can also do two special tasks with menu cells: re-sequencing and assigning Command keys. By re-sequencing, you change the order in which cells are listed in a menu (see Figure 3-55). By assigning a Command key to a cell, you give the user of your application a command key equivalent—a shortcut way to invoke the command (as Command-x is a shortcut for invoking the Edit menu's Cut command) (see Figure 3-56).

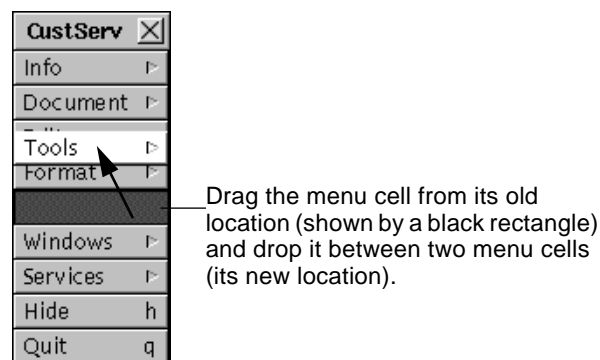


Figure 3-55 Resequencing Menu Cells

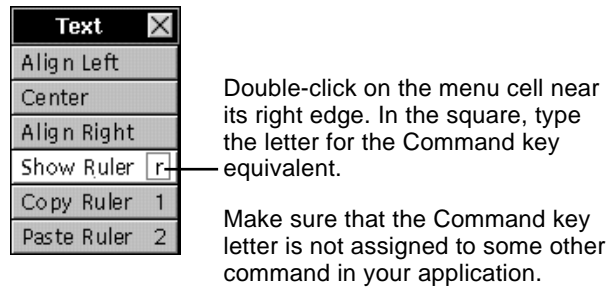


Figure 3-56 Assigning a Command Key Equivalent

Custom Menus

In addition to the standard menu commands and submenus, Interface Builder makes it easy for you to compose your own custom menus. Use the Submenu cell in the Menu palette to create custom submenus (see Figure 3-57) and use the Item cell for custom menu commands. The Print command is frequently added as a custom cell.

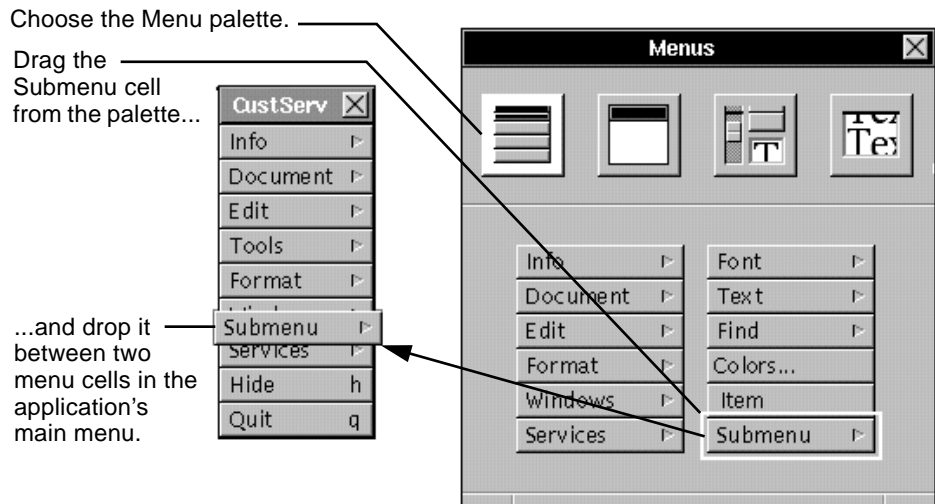


Figure 3-57 Using the Submenu Cell to Create a Custom Submenu

To create a custom submenu, do the following:

- 1. Change the title of Submenu and click on the cell to expand it.**
- 2. Then add Item cells from the Menu palette to the new submenu and change their titles.**

You can make menu cells active or inactive by default. Select the cell and set the Disabled button in the Attributes display of the cell's Inspector. See "Setting Object Attributes" for more information on using the Inspector panel.

See "Making and Managing Connections" on page 3-109 tolls with the objects that are to handle menu commands.

Setting Object Attributes

All objects on Interface Builder's standard palettes have attributes that you can initialize through the Inspector panel. This section describes many of those attributes, especially the effect they have on appearance and behavior.

The Attributes display of the Inspector panel lets you set the selected object's basic characteristics. For example, Figure 3-58 shows the Attributes display of the `NSButton` Inspector.

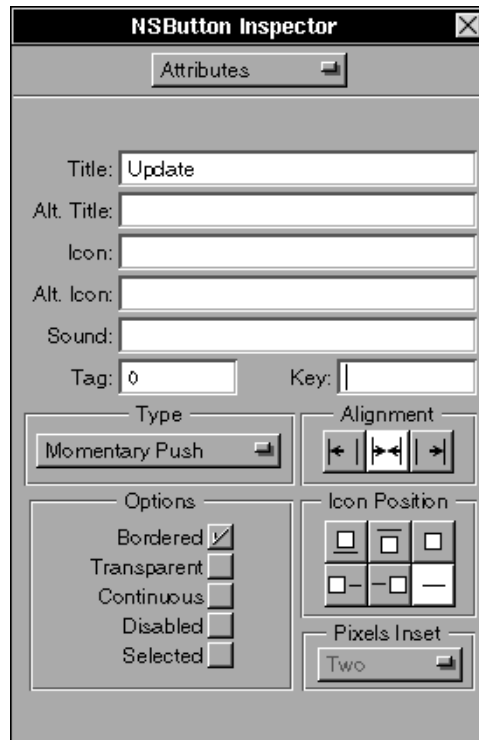


Figure 3-58 Attributes Display of NSButton Inspector

Conceptually, each of the characteristics in the Attributes display corresponds to an Objective C message to which the selected object responds. For example, Table 3-1 on page 3-65 shows the correspondence between some of the attributes displayed in the illustration above and messages that an NSButton object understands.

Table 3-1 Object Attributes and Messages

Attribute	Message
Title	setTitle:
Alt. Title	setAlternateTitle:
Icon	setImage:
Alt. Icon	setAlternateImage:
Sound	setSound:
Type	setType:
Icon Position	setImagePosition:

The Attributes displays for the selected OpenStep classes are discussed in the following sections. If you have questions about any of the attributes displayed for a selected object, consult the class specification for that object in *OpenStep Programming Reference*.

The Attributes display for the File's Owner, for Custom `NSViews`, and for custom objects that you instantiate in the nib file window lets you set the class of these objects.

Examining an Object's Attributes

You can examine the attributes of any object, whether that object is a graphical object such as a button or panel, or a non-UI object in the Instances display. To examine an object's attributes, do the following:

- 1. Select an object in the interface.**
- 2. Choose Inspector from the Tools menu.**

Note – You can also bring up the Attributes display of the Inspector panel by pressing Command-1.

The attributes for the selected object are then displayed in the Inspector panel.

Once the Inspector panel is visible on the screen, it stays there until you close it. As you select different objects, their attributes are displayed (or dimensions or connections or help links—whatever Inspector display is current).

You can also select objects in the Instances display and examine their attributes. Some of these objects (like First Responder) have no attributes. Others, like an instance of a custom class, have only one attribute.

Click on an object in the Instances display to select it, as shown in Figure 3-59.

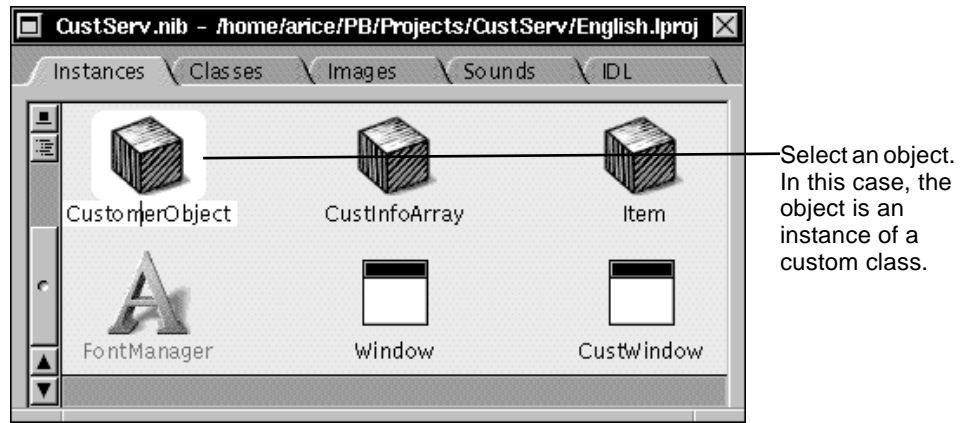


Figure 3-59 Selecting an Object in the Instances Display

If the Inspector panel is not visible, choose Tools Inspector or press Command-1. As before, the attributes for the selected object are displayed (see Figure 3-60).

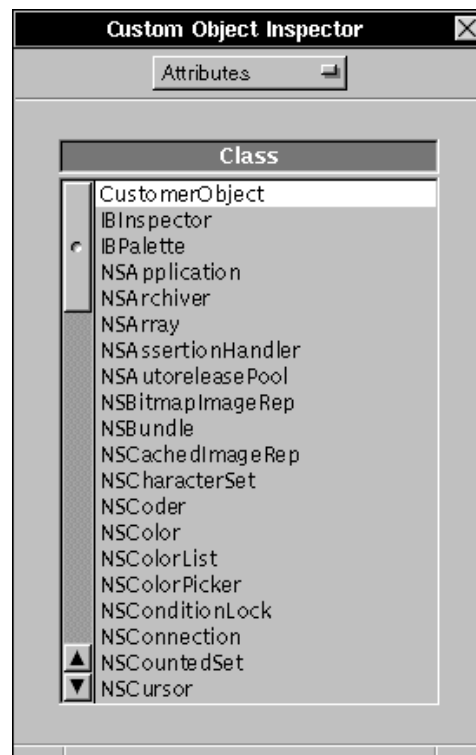


Figure 3-60 Attributes Display for a Custom Class

Customizing Windows and Panels

You can customize windows and panels in the following ways:

- Set the window title.
- Determine how the Window Server buffers window contents.
- Choose the window's controls.
- Set the window's options.

A single Attributes display of the Inspector panel serves for both windows and panels, as shown in Figure 3-61.

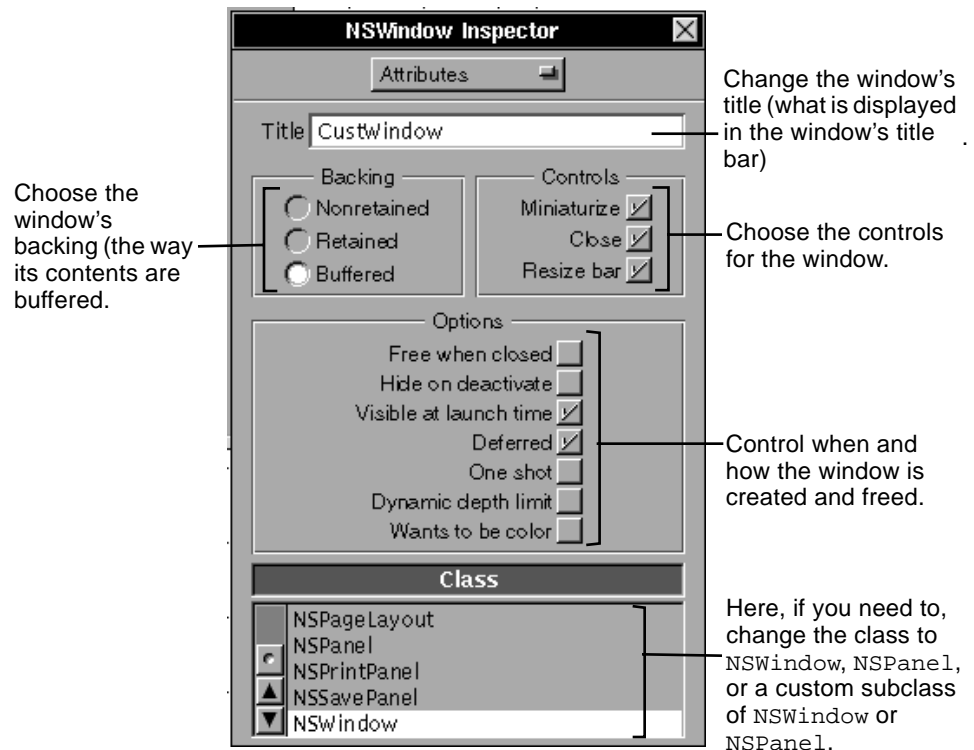


Figure 3-61 Attributes Display for Windows and Panels

Window Backing

When a window is partially covered by another window and then is re-exposed, its backing—its type of backup buffer—determines how the system re-draws the exposed part.

- **Nonretained:** The application is responsible for all drawing on the screen because there is no buffer. If the application does not do anything when the window is uncovered, the re-exposed part is replaced by the background color. Nonretained windows are appropriate for transitory images that you do not need to save.

- **Retained:** Pixels are copied to the buffer for all covered parts of the window. When an obscured part of the window is later revealed, only that part of the window is redrawn—using the contents of the buffer—the rest of the window is not redrawn. A retained window is the appropriate choice for most situations.
- **Buffered:** The window is drawn first in the buffer and then copied to the screen. When an obscured part of the window is revealed, the entire window is refreshed using the buffer. A buffered window is appropriate when you do not want users watching complicated images being rendered on-screen. It is also the best choice for animation or for redrawing lines of rapidly typed text.

Changing Class and Custom Windows

In the Class section of the Attributes display, you can change the class of a window to `NSWindow`, `NSPanel`, or any custom subclass of `NSWindow`.

Window Controls

You can choose that the window or panel have any of the three controls shown in Figure 3-62.

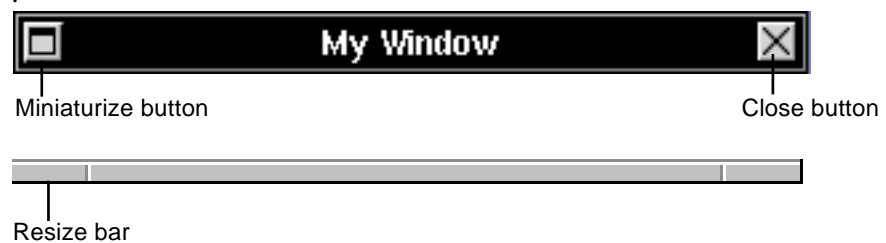


Figure 3-62 Window Controls

Window Options

You can select any of the window options listed in Table 3-2.

Table 3-2 Window Options

Option	Description
Free when closed	The window is to be freed when it is closed.
Hide on deactivate	The window should disappear when the application is deactivated.
Visible at launch time	The window should appear when the application is launched.
Deferred	A window device for this object is deferred until it is placed on-screen.
One shot	The window device is freed when the window is removed from the screen.
Dynamic depth limit	The window's depth limit can change to match the depth of the screen.
Wants to be color	The window is displayed on a color screen (two-monitor system only).

What is the Difference Between a Window and a Panel?

A panel is a window that serves an auxiliary function within an application. Because it is intended for a supporting role, a panel typically has these features:

- A panel can be the key window, but never the main window.
- When the application is deactivated, the panel moves off-screen (it is removed from the screen list). When the application is reactivated, the panel is displayed again.
- When a panel is closed, it moves off-screen; it is not destroyed.
- Unlike a window, a panel forwards Command key-down events (like carriage returns) to its views.
- When instantiated programmatically, panels have a grey background by default, while programmatically created windows have a white background.

Also, a panel usually has fewer controls: sometimes only a close button; rarely a resize bar; and sometimes no controls at all.

You can make some panels exhibit the following types of special behavior for specialized roles:

- A panel can be precluded from becoming the key window until the user makes a selection in it.
- Some panels (for example, palettes) can float above windows and other panels.
- You can have a panel receive mouse and keyboard events while an attention panel is on-screen. Actions within the panel can thus affect the attention panel.

Setting Button Attributes

The Attributes display for buttons enables you to set a button's type, title, icon, alternate title and icon, and various other characteristics. The object labeled Button on the Views palette is only one style of button (albeit the most common style). The palette also holds radio buttons and switch buttons. Using the Attributes display for buttons (see Figure 3-63), you can customize any palette button, making it something that is uniquely suitable for a particular circumstance.

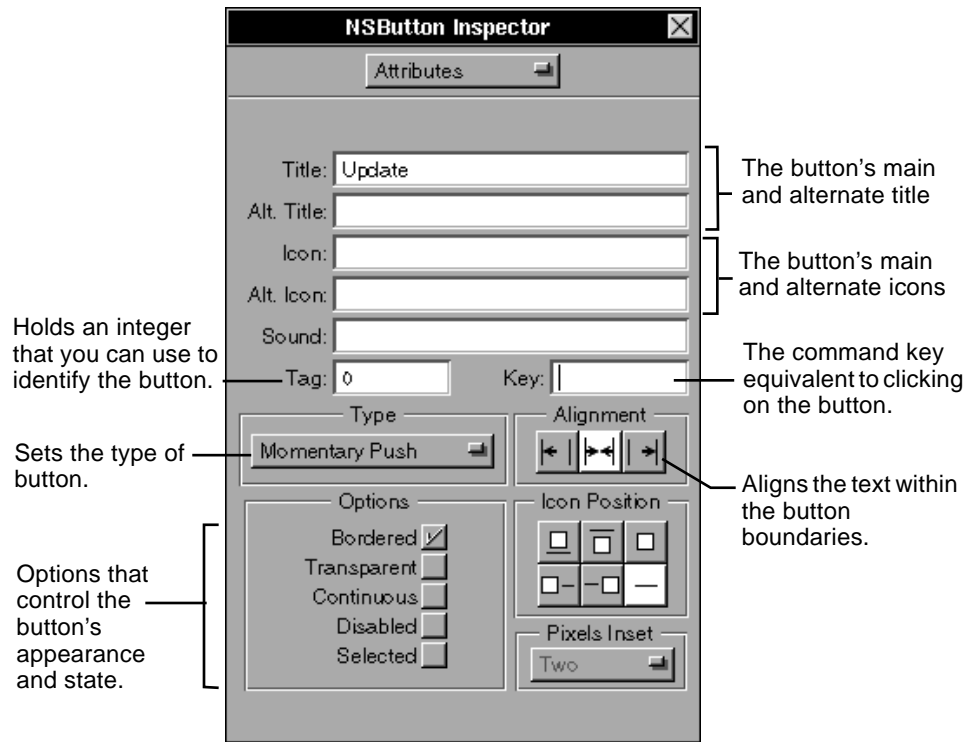


Figure 3-63 NSButton Attributes Display

The Anatomy of a Button

A button is essentially a two-state `NSControl` object. When a user clicks on a button, an action message is sent to a target object. It is two-state because it is either on or off, and when it is on, it typically sends its action message. For a button, the states are also known as "normal" (off) and "alternate" (on).

You can set a button's attributes so that the normal and alternate states show different images, or so that in the alternate state the button is highlighted. You can configure a button to send its message continuously rather than discretely. You can associate a command key-equivalent with a button.

A button is actually a compound object: a `NSButton` object and a `NSButtonCell` object. Because they inherit from `NSActionCell`, `NSButtonCells` hold the target outlet and the action message to be sent to it. By being an `NSCell` subclass, they also carry the state value of a button. `NSButtonCells` also draw the text and image of a button.

Most of `NSButton`'s methods match identically declared methods in `NSButtonCell`. Aside from dispatching the action message, `NSButton`'s unique role is to set the font of the key equivalent, and to manage the highlighting or depiction of the `NSButton`'s current state.

The Icon Position and Pixels Inset controls as well as the Sound and Icon fields are described in detail in “Associating Sounds and Images with Buttons” on page 3-75. For more information on the Tag field, see “Using Tags” on page 3-107.

You might think of storing specially configured buttons on a dynamic palette. See “Adding Custom Palettes, Inspectors, and Editors” on page 3-169.

Titles and Icons

The text in the Title field is what is displayed in most buttons; this is the text you can select by double-clicking inside the button. The name in Icon identifies an image stored in the nib file (Images display of the nib file window) that is displayed within the button. The alternate title (Alt. Text) and the alternate icon (Alt. Icon) appear when the user clicks on a button of type Momentary Change or Toggle.

Key Equivalent

The Key field identifies a keyboard alternative to clicking the button. Possible values are: `\b` (Backspace), `\d` (Delete), `\e` (Escape), `\t` (Tab), and `\r` (Return).

Button Type

A button can be of any of the types listed in Table 3-3.

Table 3-3 Button Types

Type	Button Behavior When Clicked
Momentary Push	Button is highlighted, appears to be pressed.
Momentary Change	Alternate button title and icon appear (as long as mouse button held down).
Momentary Light	Button is highlighted, but no illusion of being pressed.
Push On/Push Off	Click once and button is highlighted with illusion of being pushed in; click again and it returns to normal.
On/Off	Click once and button is highlighted; click again and it returns to normal.
Toggle	Alternate button title and icon appear; click again for main title and icon.

Button Options

You can select any of the button options listed in Table 3-4.

Table 3-4 Button Options

Option	Description
Bordered	A line is drawn around the button's border.
Transparent	The button has no border, text, icon, or background color.
Continuous	The button sends its action message continuously when pressed.
Disabled	Prevents activation of the button; text is in gray.
Selected	The button, when initialized, is to be selected (switch and radio buttons).

Associating Sounds and Images with Buttons

To associate a sound or image with a button, do one of the following:

- Drag the icon representing a sound or image from the nib file window (see Figure 3-64) or the Workspace (see Figure 3-65 on page 3-76) and drop it over a button.
- Enter the file name of the sound or image in the appropriate field of the button's Attribute's display.

You associate a sound or image with a button by dragging the icon representing the resource and dropping it over the button.

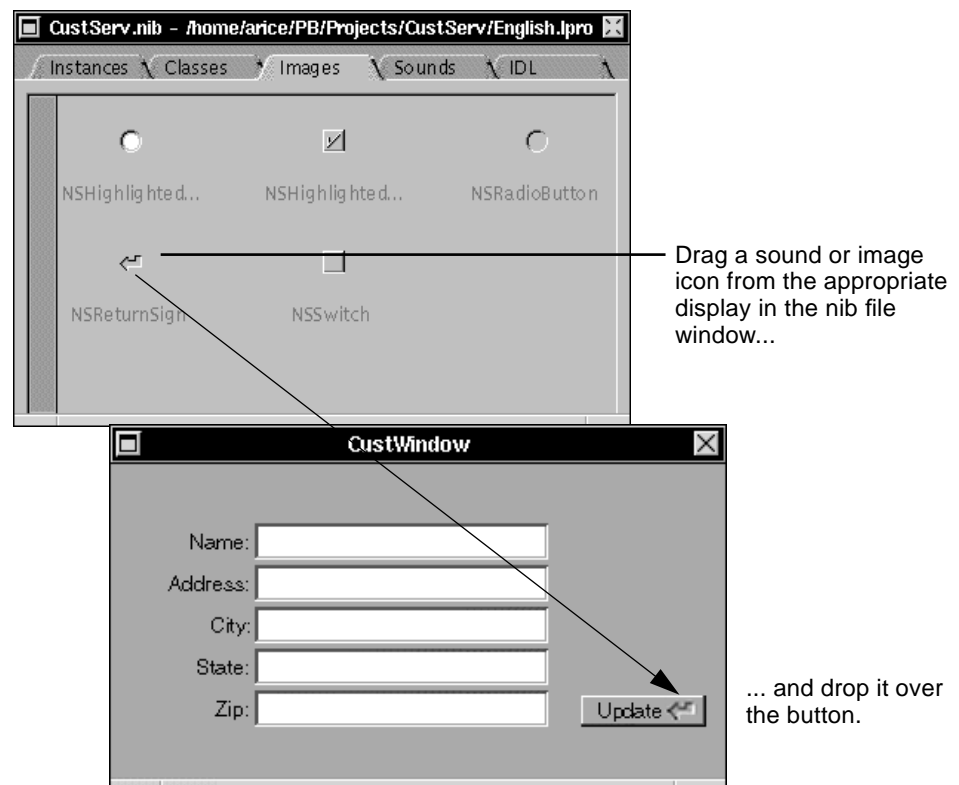


Figure 3-64 Associating an Image with a Button

When you click on a button that has a sound associated with it, it plays the sound. Images appear in buttons with or without text. They can play more than an iconic or decorative role; when you drag the `NSReturnSign` image onto a button, for instance, the carriage return key-equivalent is associated with the button ("`\r`" in the Key field).

You can drag sounds and images from the Workspace Manager's File Viewer too, as shown in Figure 3-65.

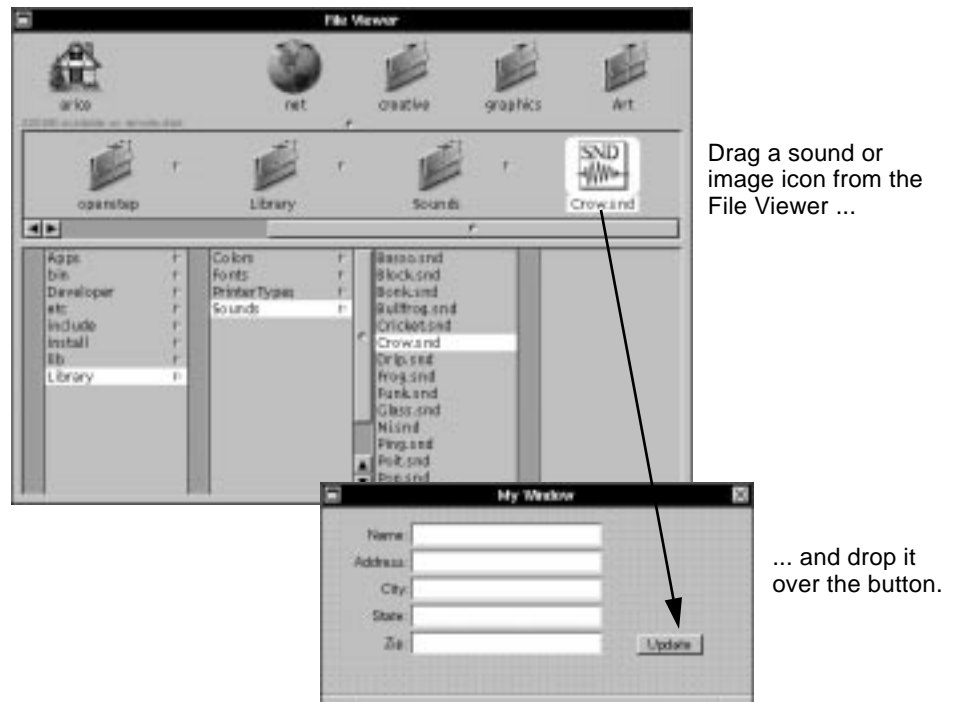


Figure 3-65 Associating a Sound with a Button

The sound or image automatically is added to the Sound or Image section of the nib file window.

Several fields and controls in the Inspector's Attributes display for buttons relate to images and sounds (see Figure 3-66 on page 3-77).

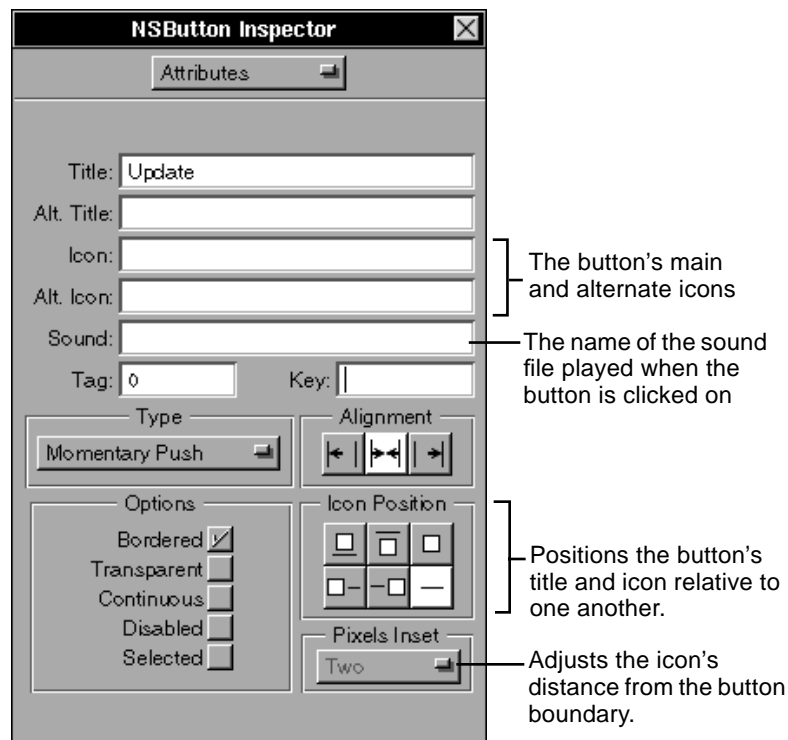


Figure 3-66 NSButton Attributes that Relate to Sounds or Images

Note that the name of a sound or image in this display is the file name (blat.snd and NSReturnsign.tiff, for example) minus the extension. Instead of dragging and dropping sound and image icons, you can type their file names (minus the extension) in the appropriate field. However, you should insert the resource into the nib file or the project before typing its name in the Attributes display.

Note – For most situations, the recommended course of action is to add sounds and images to your project. If you add resources only to a nib file, those resources might not be available to an application unless the nib file has been loaded.

The six buttons in the Icon Position group position the button title and icon relative to each other. Thus you can have the title above, below, to the left, or to the right of the icon, or show only the title, or show only the icon. The Pixels Inset pull-down menu gives several pixel distances for adjusting the spacing between the icon and the nearest edge of the button.

Note – If you want to import images into your interface for decorative purposes, drop the image on a button (if smaller than the image, the button will resize to contain it). Position the button, then deselect the Bordered option and select the Transparent and Disabled options.

Managing Sounds and Images

You can manage sounds and images in your interface in the following ways:

- Add an image or file by dragging the file icon representing the resource from the File Viewer and dropping it over the nib file window.
- Examine the sound or image in the Inspector's Attributes display.

As shown in “Associating Sounds and Images with Buttons” on page 3-75, you can add sounds and images to a nib file as a side effect of associating those resources with a button. You can also insert sounds and images into a nib file by dragging them from the File Viewer and dropping them over the nib file window, as shown in Figure 3-67.

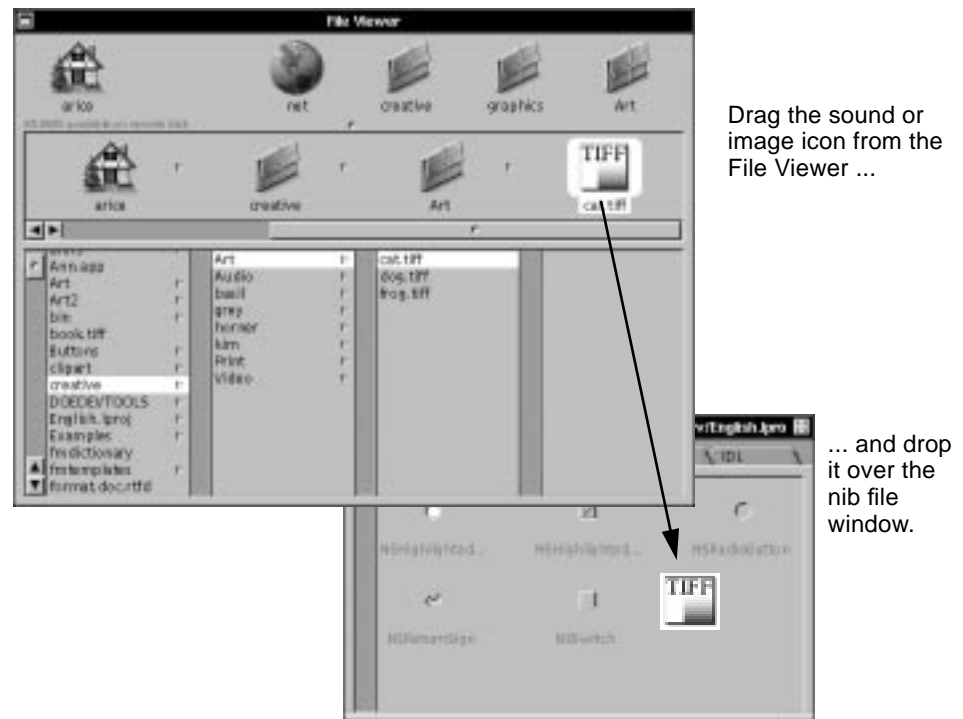


Figure 3-67 Adding a Sound or Image to a Nib File

You can drop a sound or image over the nib file window no matter what display is showing. The sound or image file will be correctly added to the appropriate display.

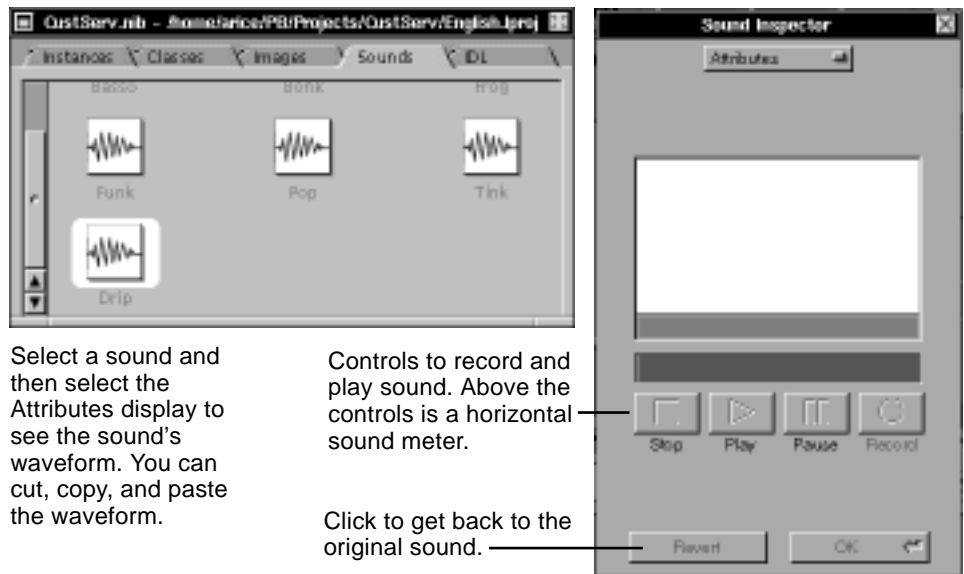
Although the association of sounds and images with buttons is an important reason for putting them into a nib file, there are other reasons. When you play a sound or composite an image in your code, the search path (if you supply no path) starts with the application's executable (already loaded resources), the main bundle, and the main bundle's `.lproj` directories. Then the following paths are searched:

- the appropriate subdirectory of the user's `~/openstep/Library` directory
- the appropriate directory in `/usr/local/openstep/Library`
- the Application Kit

If you do not want to risk a sound or image not being in one of these standard directories, then you should store it in the a nib file or in the project.

Sounds and images have their own Attributes displays. If the Inspector panel is not visible, access it as you do for palette objects: choose the Inspector command from the Tools menu or press Command-1.

Figure 3-68 explains the Attributes display for sounds.



Select a sound and then select the Attributes display to see the sound's waveform. You can cut, copy, and paste the waveform.

Controls to record and play sound. Above the controls is a horizontal sound meter.

Click to get back to the original sound.

Figure 3-68 Inspecting Sound Attributes

If your system has a microphone or some other input source connected, you can record new sounds. Click on OK to save new sounds.

For images, the Attributes display (see Figure 3-69) is mostly useful for images that are too large to show in the Images display of the nib file window.

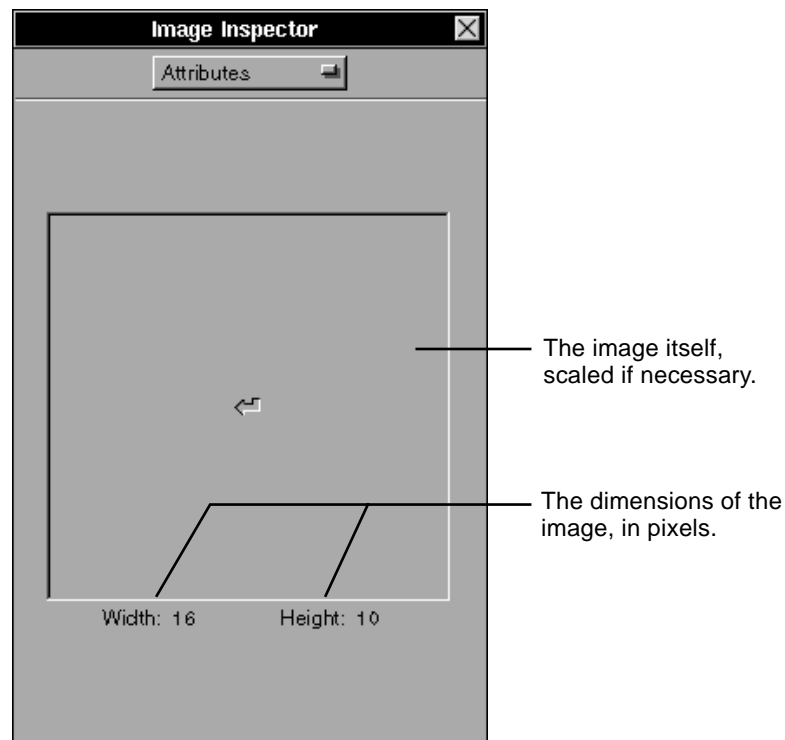


Figure 3-69 Inspecting Image Attributes

Customizing Titles, Text Fields, and Scroll Views

For `NSTitles`, `NSTextField`s, and `NSScrollView`s, you can set background and text color, text alignment, border style, tag, and options affecting access to text.

OpenStep gives you several ways to display, format, and control access to text. The Attributes displays for `NSTitles`, `NSTextField`s, and `NSScrollView`s have controls for initializing those objects with various characteristics. A title is a specially configured text field: non-selectable with transparent backgrounds and no borders.

To see what certain effects look like, drag an `NSTextField` onto a window and click on the buttons on the display shown in Figure 3-70 on page 3-82.

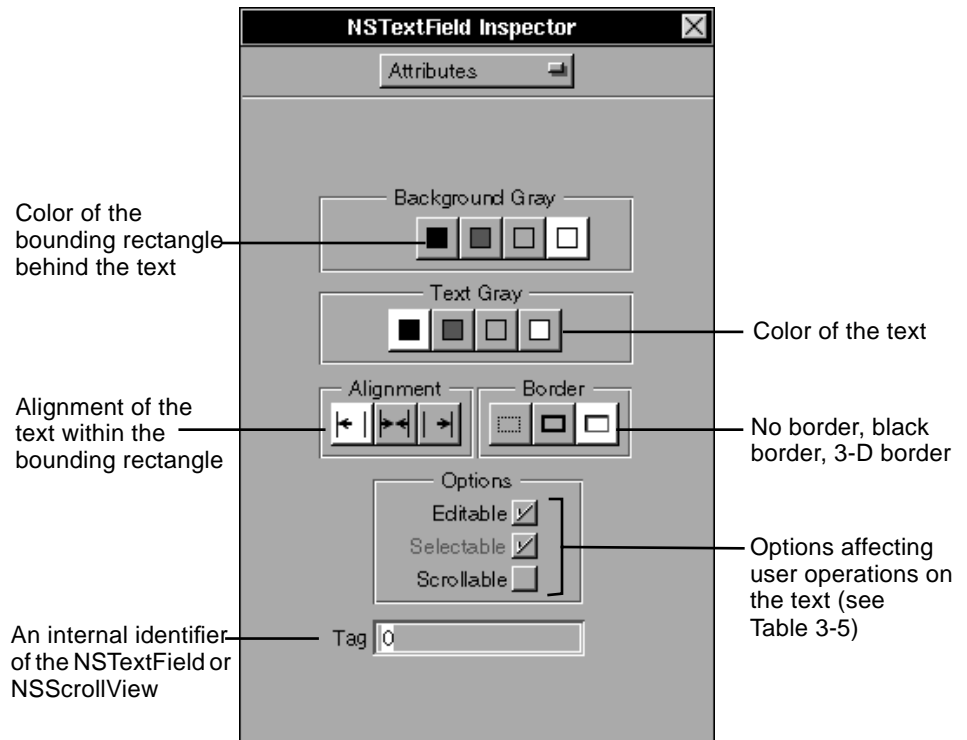


Figure 3-70 Setting NSTextField Attributes

An NSScrollView object is a compound object consisting of one or two NSScroller objects and a NSClipView object, which has as its document view (subview) an NSText object in Interface Builder. The document view is what is scrolled. The NSScrollView object has a slightly different Attributes display (see Figure 3-71): no text alignment buttons and a different set of options, which are listed in Table 3-5.

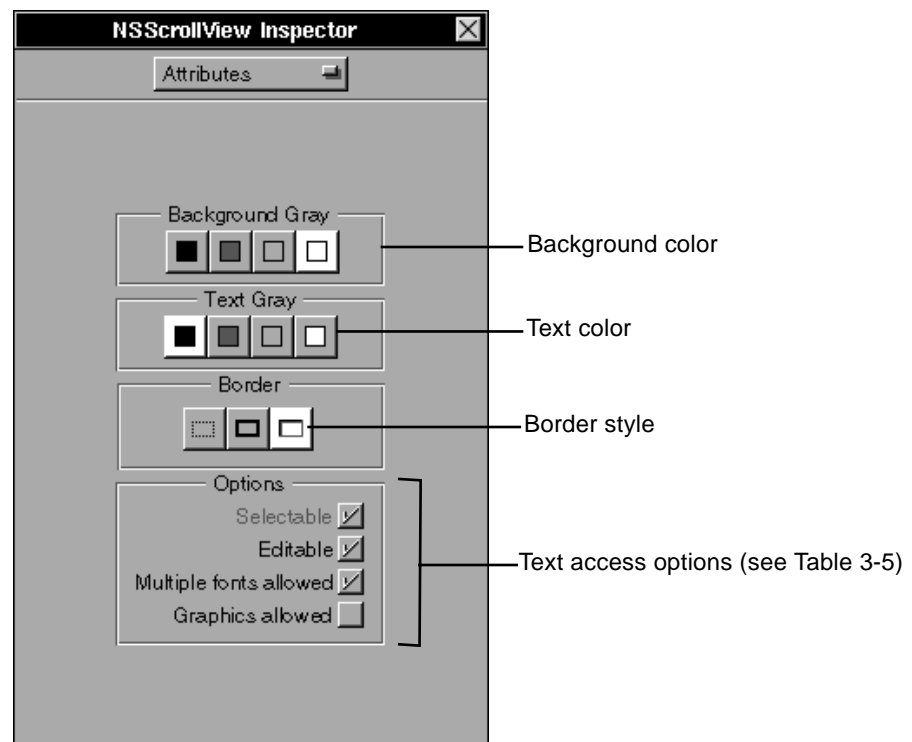


Figure 3-71 Setting NSScrollView Attributes

Table 3-5 Title, Text Field, and Scroller Options

Option	Description
Editable	Set to permit the user to edit text.
Selectable	Set to permit the user to select text.
Scrollable	Set to enable typing beyond the borders of the field (text scrolls to the left—NSTextField only).
Multiple fonts allowed	Set to put text in RTF format. (NSScrollView only).
Graphics allowed	Set to put text in RTFD format (graphics can be inserted—NSScrollView only).

Note – A tag is an internal identifier of an object that you can use in your code. See “Using Tags” on page 3-107 for more information.

Note – For more information on the `NSTextField`, `NSScrollView`, `NSScroller`, `NSText`, and `NSClipView` classes, see the appropriate specification in *OpenStep Programming Reference*.

Setting Textual Attributes

You can set the following textual attributes:

- The font characteristics of selected text
- The alignment of selected text within its boundaries

`NSTextFields` and `NSScrollViews` (with `NSText` objects as document views) are not the only palette objects that can contain text. Almost all palette objects—from buttons to browsers—can display text. You can set the font and alignment attributes of this text as shown in Figure 3-72.

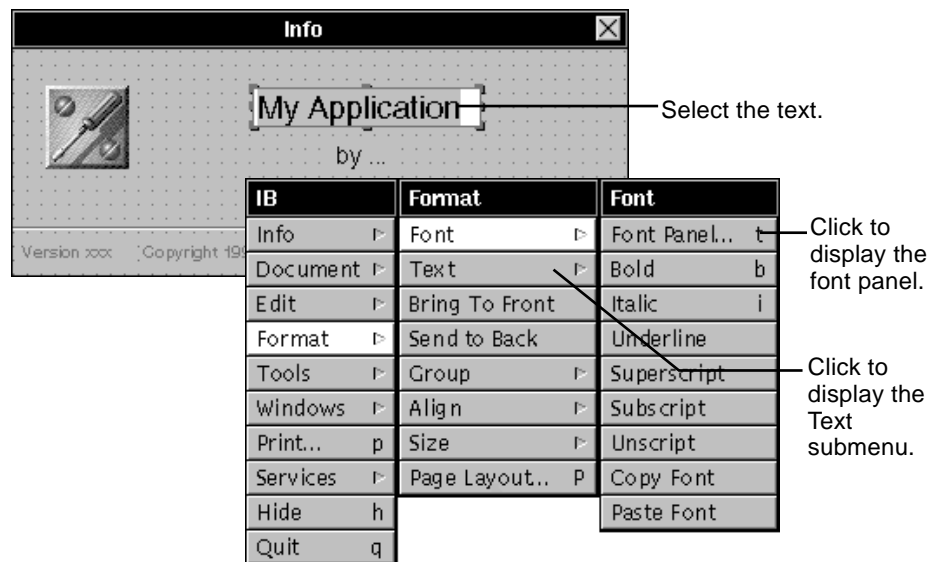


Figure 3-72 Setting the Attributes of Text

The Text submenu of the Format menu also has commands that affect selected text; it offers options for aligning text and for displaying, copying, and pasting the ruler in an NSText object. With rulers you can set tabs and indentation. Note that rulers can only appear in NSText objects (for instance, inside a scroll view).

If you choose Font Panel from the menu (or press Command-t), the Font Panel (shown in Figure 3-73) is displayed.

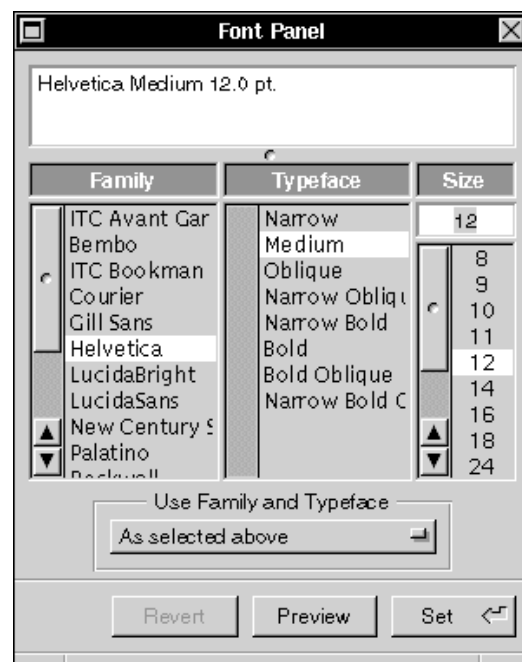


Figure 3-73 Font Panel

You can use the standard menus for setting font and alignment attributes for any text in your interface. See *Using the OpenStep Desktop* and online help for complete information.

Setting Box (Group) Attributes

You can set title position, border style, and horizontal and vertical offsets for an NSBox object.

When you group a selection of objects, a box encloses that object. The box (actually the box's content view) becomes the superview of the enclosed objects. If you select the box in Interface Builder, you can move, copy, paste, and delete the group of objects as one.

The box has several attributes that you can set, as shown in Figure 3-74.

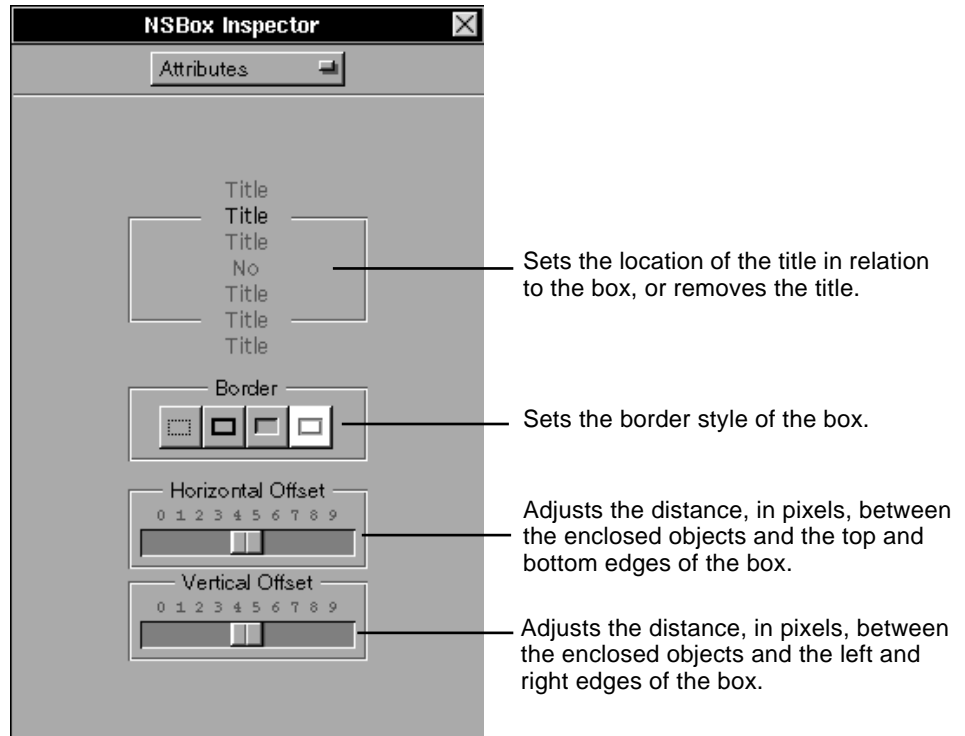


Figure 3-74 Setting NSBox Attributes

You can also create a box object in your interface by dragging the `NSBox` object from the Views palette. To group objects within this box, double-click on the box when it is on the interface. Then drag objects from a palette and drop them within the box. Double-click on the box again when you are finished.

You can drag a box onto your interface and then programmatically replace its content view (blank by default) with another `NSView` object, or programmatically add subviews to the content view. You can also manipulate this box to make decorative rectangles and lines.

Note – To make a line in an interface, such as a divider line between sections of a panel, drag a box onto the interface, Then switch off the title and make the box as narrow as possible in the required dimension (vertically or horizontally). Finally, set the offset (vertical or horizontal, whichever is applicable) to zero.

Customizing Browsers

You can select the browsing controls (see Figure 3-75) and browser options (see Table 3-6) for `NSBrowser` objects.

Browsers display lists of data and allow users to select items from the list. They can hold one-dimensional lists or hierarchically organized lists of data such as directory paths. Browsers display these hierarchical levels in columns, which users can navigate using buttons or scrollers. An entry in a column can be either a leaf node or a branch node. Leaf nodes terminate a path; branch nodes, which have a right-arrow icon, lead into the next level in the hierarchy.

Browsers have attributes affecting their navigation controls, methods of selection, and appearance, as shown in Figure 3-75.

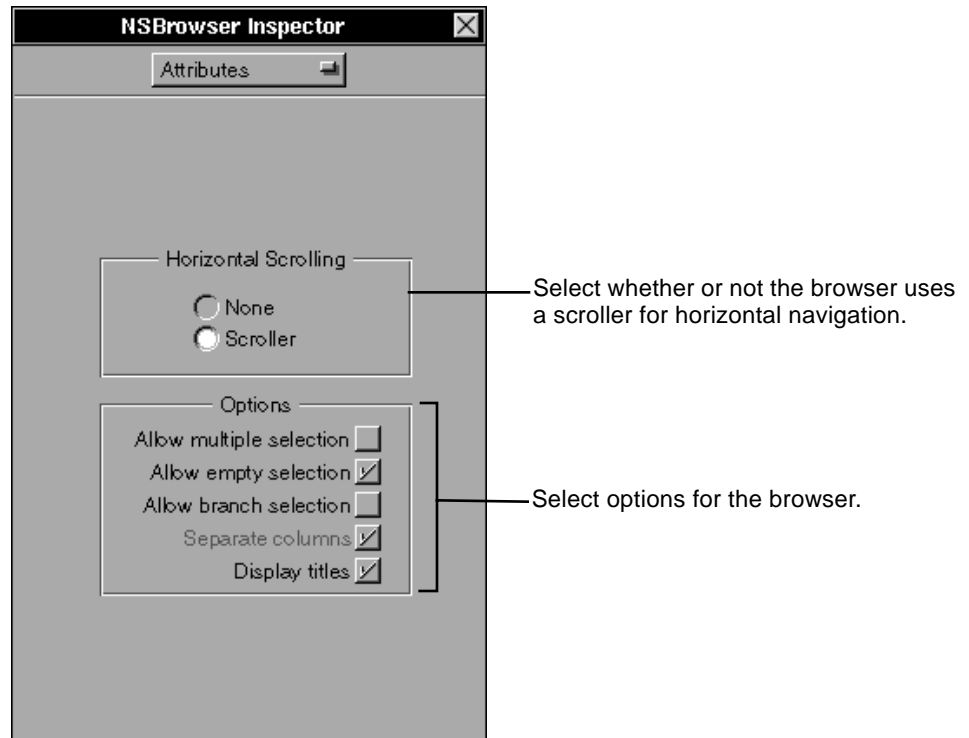


Figure 3-75 Setting NSBrowser Attributes

The Horizontal Scrolling buttons determine whether the browser uses a scroller for navigating levels. Click on these radio buttons to see what the scrolling controls look like.

A browser automatically uses a scroller for vertical navigation of lists.

Table 3-6 Browser Options

Option	Description
Allow multiple selection	Permits the selection of more than one node at a time.
Allow empty selection	Makes it possible to have no cells selected; otherwise, first cell in column is selected by default.

Table 3-6 Browser Options

Option	Description
Allow branch selection	Permits the selection of branch nodes (such as directories).
Separate columns	Separate columns by a beveled bar (if not set, a black line is displayed).
Display titles	Titles are above columns and column divider is beveled bar.

Setting Attributes of Menu Cells and Pop-up Buttons

You can set whether the list is a pop-up or pull-down type (not applicable to menu cells). You can set whether the cell is initially disabled. And you can assign a tag to the cell.

Menus, pop-up lists, and pull-down lists are compound objects that contain `NSMenuItem` objects. The Attributes displays for menu cells and pop-up/pull-down lists are almost identical. Figure 3-76 shows the display for pop-up/pull-down lists.

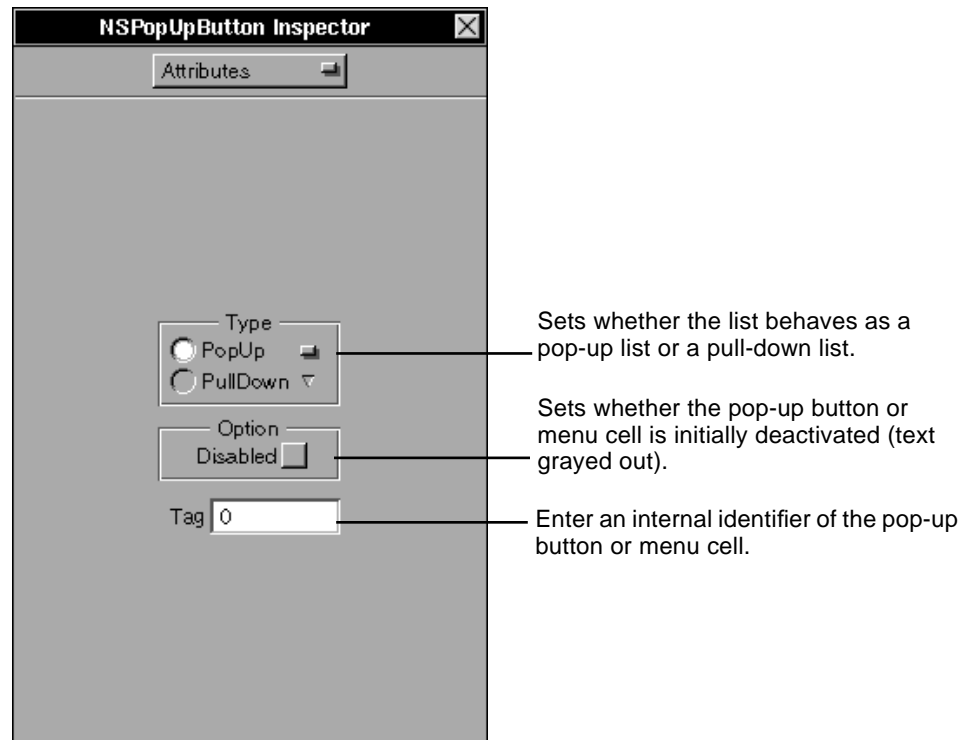


Figure 3-76 Setting NSPopUpButton Attributes

If you disable a menu cell in Interface Builder, its text is gray when the application is launched. When the user clicks the cell, no action message is sent. If conditions change to make the cell's function relevant, your code must re-enable the cell.

Pop-Up Lists and Pull-Down Lists

The object `Item1` on Interface Builder's Views palette is actually a trigger button (see Figure 3-77) whose target is an `NSPopUpButton` object. When you double-click on the trigger button on your interface, three menu cells appear; you can initialize their titles or (in the Attributes display) disable them and assign them tags. In a running application, the `NSPopUpButton` object, once triggered, tracks the mouse until the user releases it, at which time it sends the selected action message to its target and disappears.

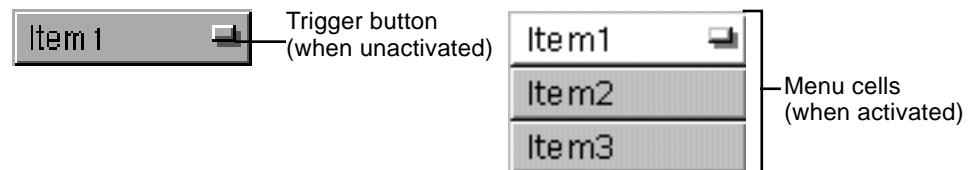


Figure 3-77 Pop-up List's Trigger Button and Menu Cells

A pop-up list's trigger button always displays the item that was last selected.

In a pull-down list the trigger button's title is fixed. This type of list is effective for selecting actions in a very specific context, like the Operations pull-down list in Interface Builder's Classes display.

Note – Once you expose a pop-up list's menu cells, you can add more menu cells to it from the Menus palette (see “Creating Menus” on page 3-60).

Note – A tag is an internal identifier of an object that you can use in your code. See “Using Tags” on page 3-107 for more information.

Compound Objects

Most of the objects you can drag from the standard Interface Builder palettes are actually compound objects. They consist of two or more objects that work together in specific ways.

NSControl and NSActionCell

An `NSControl` (an instance of an `NSControl` subclass) functions as an event translator. It translates a user event like a mouse click into a action message and directs that message to another object in the application (the target).

`NSControls` supply the mechanism but not the content of the target/action paradigm. They need `NSActionCell` objects (or instances of `NSActionCell` subclasses) to hold this information:

- target - the object receiving the action message
- action - the method that specifies what the target is to do

At least one of these cells occupies the same area as its `NSControl`. Because it descends from `NSCell`, a cell also has content (text or image), which it draws upon request from its `NSControl`.

This division of responsibility makes for greater efficiency. This is especially true because an `NSControl` can have multiple cells and send a different action message to a different target for each of those `NSCells`. Since `NSCells` are lightweight objects, it is more efficient in some contexts to associate one `NSControl` with many `NSCells`.

Matrices

Instances of `NSButton`, `NSSlider`, and `NSTextField` are `NSControl` objects each of which are bound to a single `NSCell`. An `NSMatrix` (an instance of the `NSMatrix` class) is also an `NSControl`, but it manages more than one `NSCell`. It organizes its `NSCells` in rows and columns. The `NSCells` must be of the same size and usually are of the same class (although an `NSMatrix` can have instances of different subclasses of the `NSCell` class).

An `NSMatrix` allows each of its `NSCells` to have its own action and target. An `NSMatrix` also has its own action and target. If an `NSCell` does not have an action, the `NSMatrix` sends its own action to its own target. If an `NSCell` does not have a target, the `NSMatrix` sends the `NSCell`'s action to its own target.

In Interface Builder you can convert a single-celled `NSControl` object into a `NSMatrix` by Alt-dragging a resize handle of that `NSControl`. The associated `NSCell` object, whether it is an `NSButtonCell`, `NSSliderCell`, or `NSTextFieldCell`, is duplicated for each row and column of the `NSMatrix`.

`NSForms` are a special type of matrix (`NSForm` inherits from `NSMatrix`). They have special `NSCells` (instances of `NSFormCell`) that compose both the form entry fields and the titles of those fields. An `NSMenu`, though actually a descendent of `NSWindow`, depends for its behavior on the `NSMatrix` object in its content area, which is filled with `NSMenuCells`.

Special Compound Objects

Some objects on Interface Builder's standard palettes are of a more complex composition.

`NSScrollView`

The `NSScrollView` object (see Figure 3-78) coordinates the interaction between `NSScroller` objects and an `NSClipView` object to scroll a document. It consists of one or two `NSScrollers`, an `NSClipView` object, and the document view, which is generally an `NSText` object.

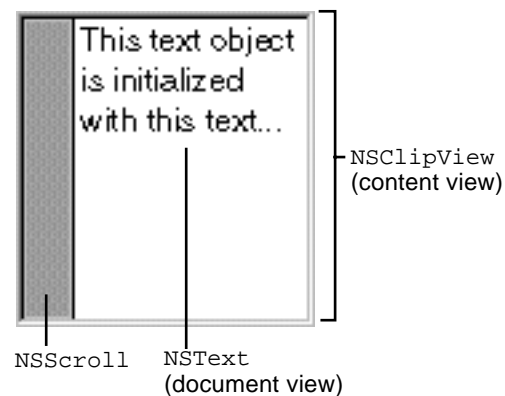


Figure 3-78 `NSScrollView`

`NSBrowser`

The `NSBrowser` object (see Figure 3-79) has scroll bars or buttons for controls, and columns to show hierarchically organized data. Each column is a `NSMatrix` of `NSBrowserCell` objects.

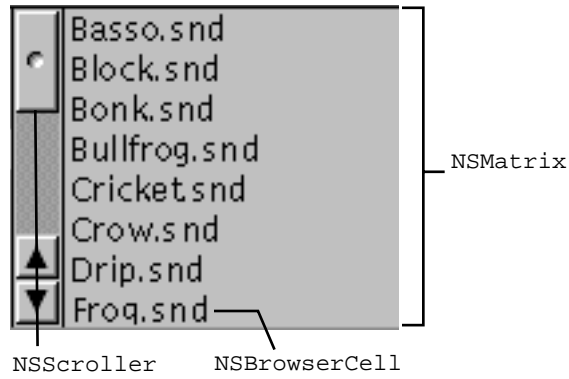


Figure 3-79 NSBrowser

NSPopUpButton

The NSPopUpButton object (see Figure 3-80) has different manifestations, depending on state. When not activated, it presents a button or button cell. When the user clicks on this trigger button, an NSMatrix of NSMenuItem objects is displayed.

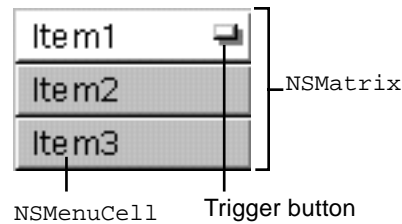


Figure 3-80 NSPopUpButton

Setting Matrix Attributes

The Attributes display for matrices, shown in Figure 3-81, allows you to determine how a matrix and its cells look and behave.

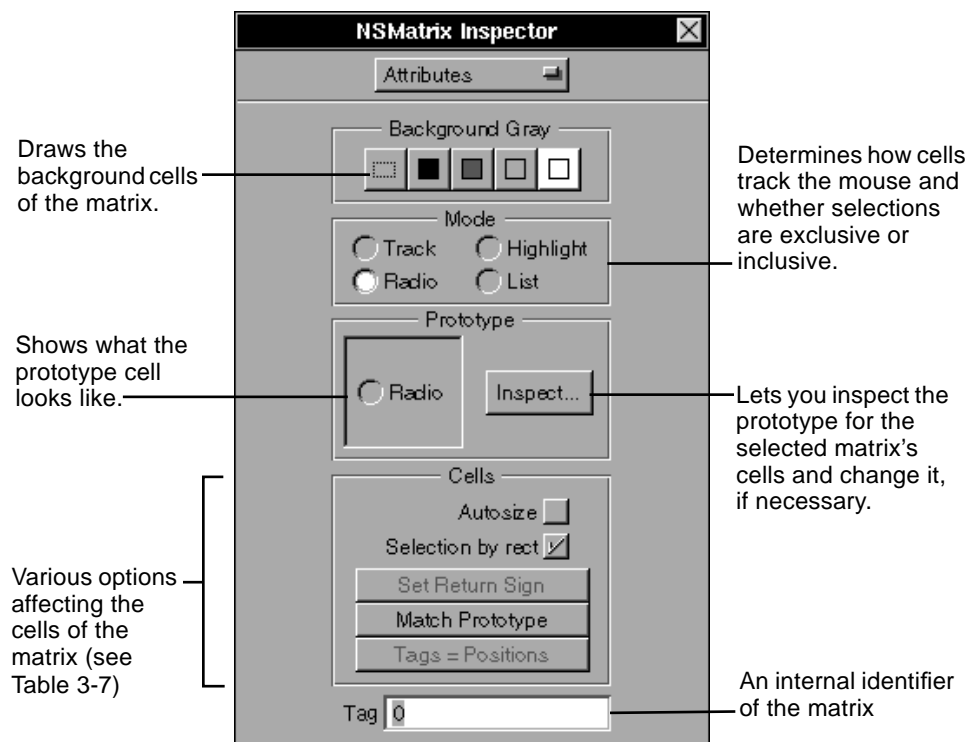


Figure 3-81 Setting NSMatrix Attributes

Matrix Selection Mode

You can set one of four selection modes to specify how cells behave when a user is dragging a mouse within a matrix. These modes also determine if selections in a matrix—a column of switch buttons, for example—are exclusive (only one allowed) or inclusive (multiple selections allowed).

- **Track:** The cells track the mouse when it is within their bounds but do not highlight themselves. This mode would be suitable for a "graphic equalizer" matrix of sliders. Moving the mouse around causes the sliders to move under the mouse.
- **Radio:** Only one cell in the matrix can be selected at a time, as is the typical case with a matrix of radio buttons.
- **Highlight:** Each cell is highlighted while it tracks the mouse, then is unhighlighted when it is done tracking. This mode allows multiple selections within a matrix. A matrix of switch buttons commonly has this mode.
- **List:** Cells are highlighted as the mouse is dragged across them, but they do not track the mouse. In this mode, a matrix supports multiple selection, enabling a user, for instance, to select a range of text.

Cell Prototype

When an `NSMatrix` object creates its cells, it typically makes them by copying a prototype cell stored as an instance variable. (It can also instantiate its cells from the cell's class.)

You can examine and alter the attributes of this prototype cell through the Attributes display. Click on the Inspect button to see the cell-prototype inspector shown in Figure 3-82.

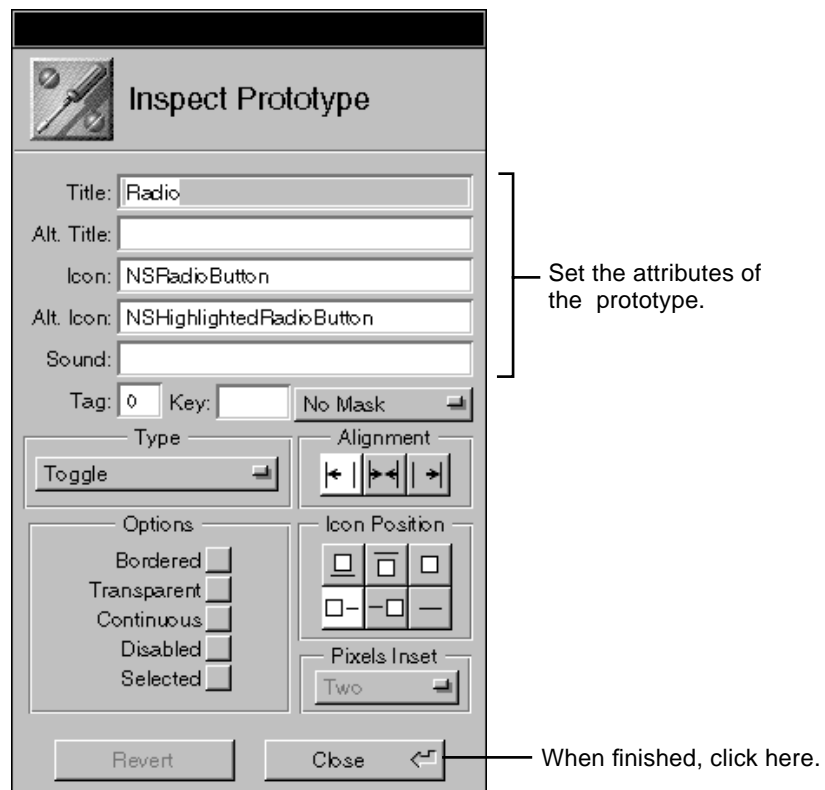


Figure 3-82 Cell Prototype Inspector

If you change the prototype, be sure to apply the changes to existing matrices by selecting them, bringing up the Attributes display, and clicking on the Match Prototype button.

Table 3-7 Cells Options

Option	Description
Autosize	If set, the cells resize themselves whenever the matrix is resized, keeping the space between cells constant. If not selected, the space between cells changes.
Selection by rect	Set to allow users to select entire rows or columns of cells by dragging the mouse.
Set Return Sign	Click to have the carriage-return equivalent set in the last button cell of a matrix. This cell has both the carriage-return icon and control key associated with it.
Match Prototype	If you have changed the cell prototype, click on this button to apply the new prototype to the selected matrix.
Tags = Position	Click on this button to resequence the tags of the cells. When you create a matrix in Interface Builder, cells are assigned tag integers starting from zero. For two dimensional matrices, the progression is from left to right (row), then down column). If you later add new cells to a matrix by Alt-dragging it to the right or down, the new cells have tag numbers of zero.

Note – A tag is an internal identifier of an object that you can use in your code. See “Using Tags” on page 3-107 for more information.

Automatically Resizing Objects

When you resize a window, the View objects in the window must often adjust their size or the distances between themselves and other objects. The Size display of the Inspector panel, shown in Figure 3-83, lets you tell a selected object how to resize itself. To set the resizing behavior of an object, do the following:

1. **Select an object.**
2. **Choose the Size display of the Inspector panel.**
3. **In the Autosizing view of the display, click on lines to make them springs or click on springs to make them lines.**

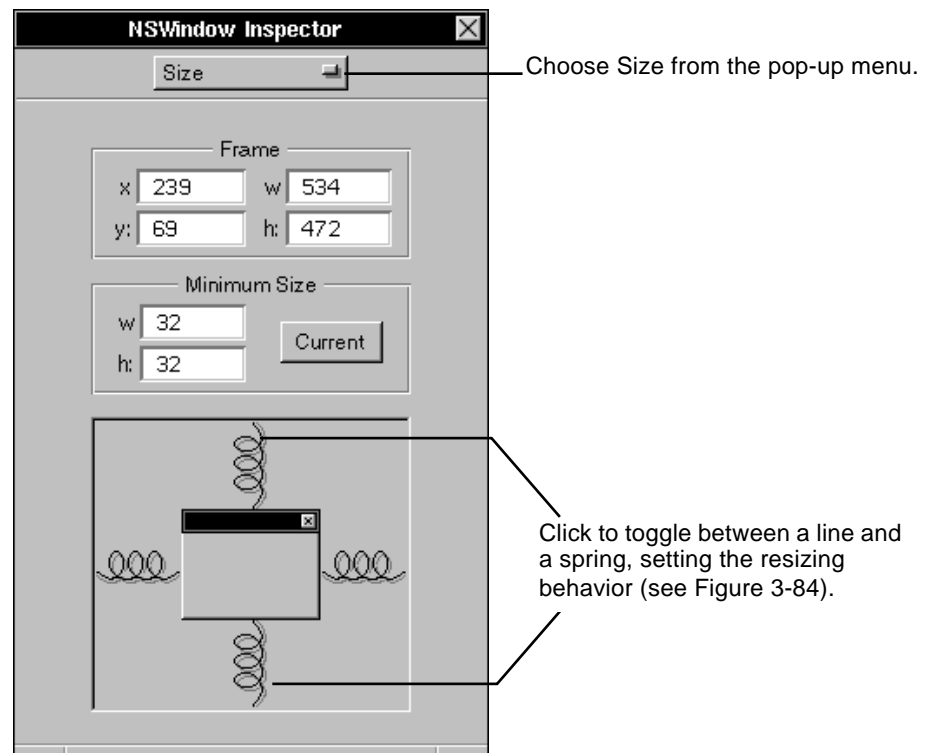
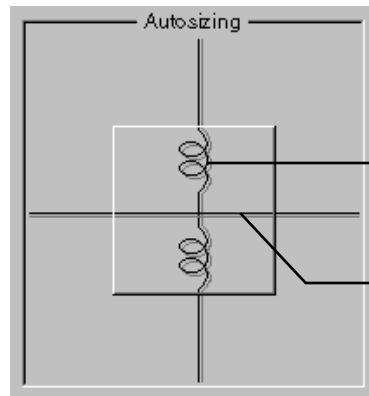


Figure 3-83 Size Inspector

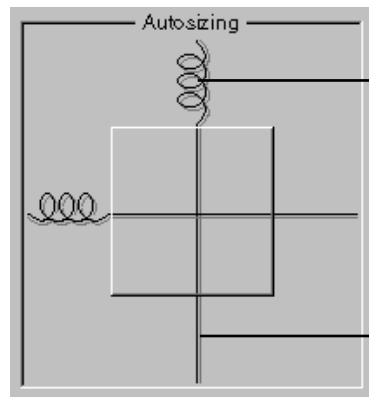
The lines inside and outside the box affect different aspects of resizing behavior, as shown in Figure 3-84.



Inside the box

The spring inside the box indicates that when the window or superview is resized vertically, the object resizes itself to maintain its distance from the top and bottom edges of the window or superview.

The straight line inside the box indicates that when the window or superview is resized horizontally, the object keeps the height at which it was initialized.



Outside the box

The spring outside the box indicates that when the window or superview is resized vertically, the space between the top edge of the object and the enclosing view or window is adjusted proportionally.

The straight line outside the box indicates that when the window or superview is resized, the object maintains the initialized distance between its right edge and the enclosing view or window edge.

Figure 3-84 Effects of Lines Inside and Outside the Autosizing Box

Note – For examples of the effects of these autosizing characteristics on views within a resized window, see “Some Effects of Automatic Resizing” on page 3-102.

If you do not make a view resize itself when its superview or window resizes, some ugly behavior could result. For instance, if the user makes a window small, objects that do not resize themselves could become truncated by the resized window's borders.

One recourse to this unwanted outcome is to specify a minimum size for the window, as shown in Figure 3-85.

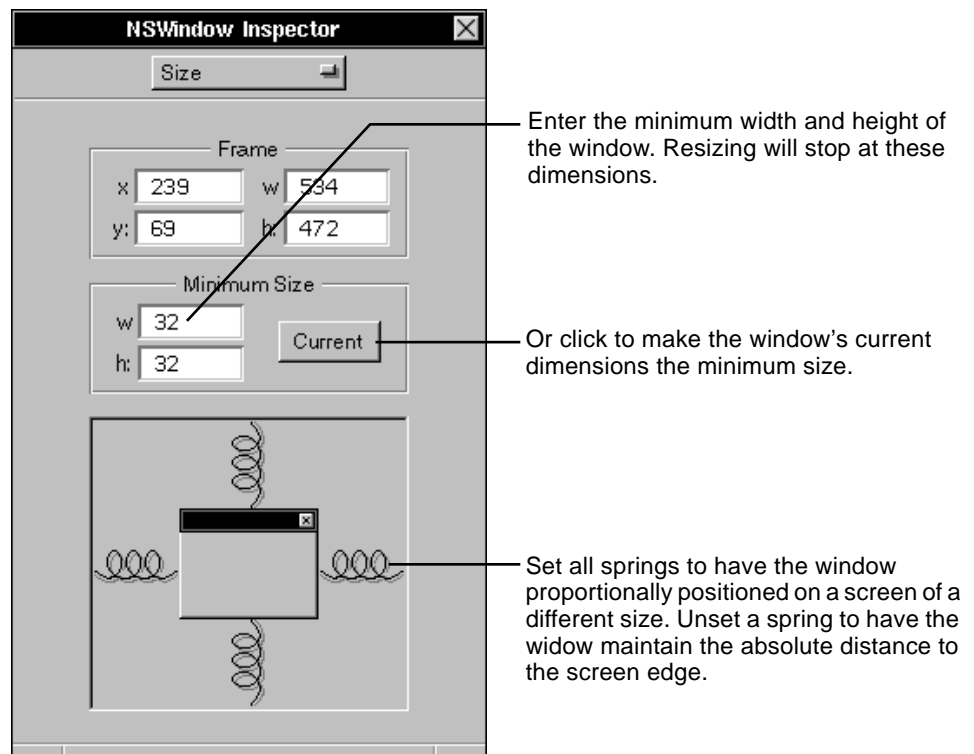


Figure 3-85 Specifying a Minimum Size for a Window

You might need to make several iterations in Interface Builder—setting resizing characteristics in objects and shrinking the window in test mode—to determine what the ideal minimum size should be.

When There Are Conflicts

You can create an impossible resizing relationship, such as specifying as fixed the object's dimensions and its distance from the window's edges. In cases of conflict, an object's fixed dimension takes precedence over its fixed distance from a border. If all dimensions are made resizable, adjustments to the window or superview's changed dimensions are made equally to the object and its distance from a border.

Note – Interface Builder includes a test mode that simulates the actual operation of the interface. In test mode, you can test the resizing behavior of your windows and views, see how connected objects communicate, play sounds associated with buttons, and do similar operations. Test mode does not test your custom objects or the connections custom objects have with the standard palette objects. See “Testing the Interface” on page 3-136 for more information.

Some Effects of Automatic Resizing

The window in Figure 3-86 has two identical scroll view objects. Different autosizing springs are set in each, and then the window is resized in test mode. Figure 3-87 and Figure 3-88 show you the results.

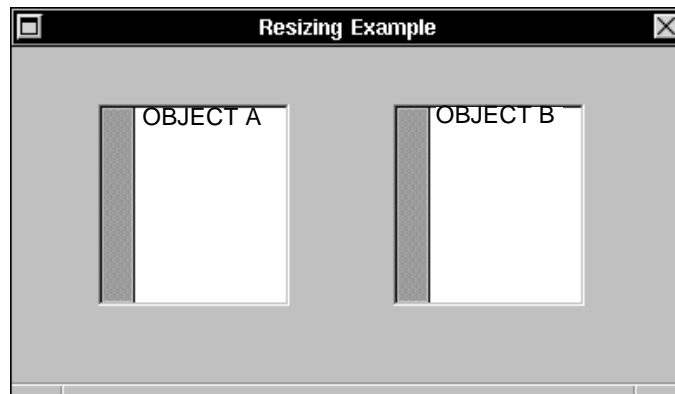


Figure 3-86 Resizing Example

In Figure 3-87, one object resizes vertically while the other does not (distances to borders are absolute for both). The result: the object that does not resize itself is truncated when the window is vertically shortened.

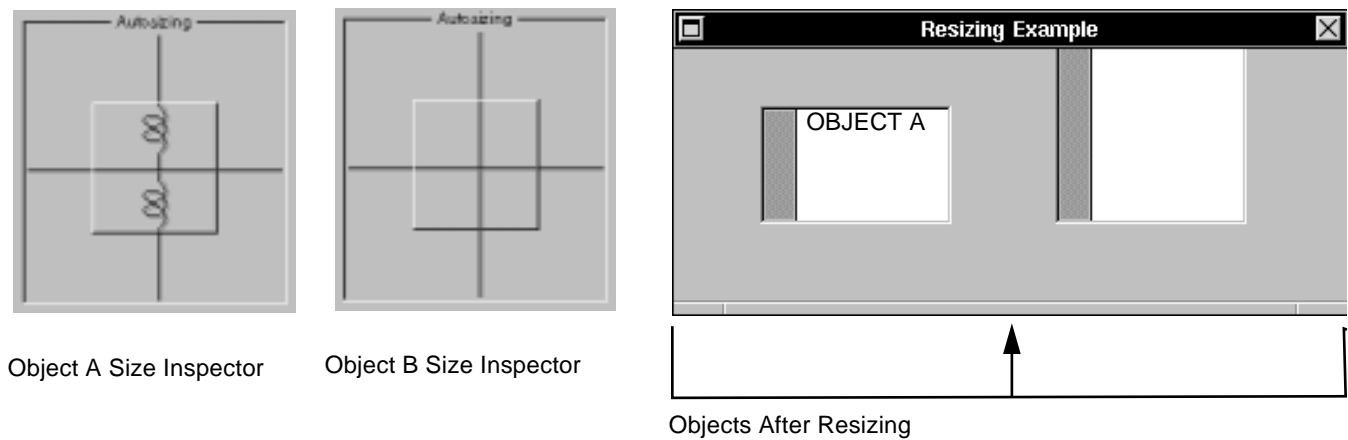


Figure 3-87 Object A Resizes, Object B Does Not

In Figure 3-88, both objects resize themselves, but Object B maintains its distance to surrounding objects. This causes Object B to be more severely resized than Object A.

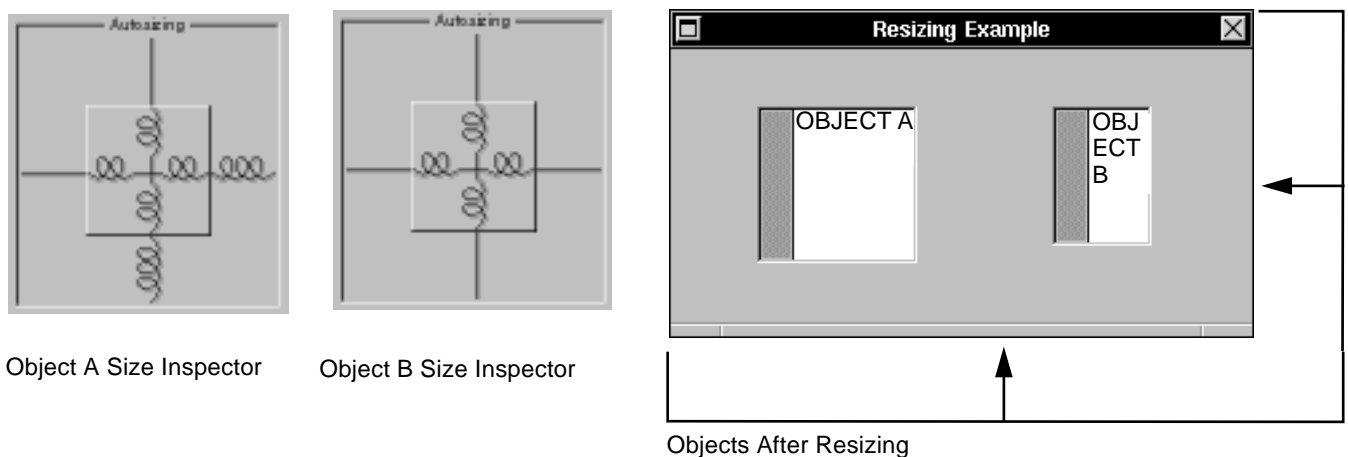


Figure 3-88 Both Object A and Object B Resize

To learn more about the effects of resizing, try some experiments on your own using different combinations of objects and autosizing attributes.

Automatic Resizing: An Example

The example interface shown in Figure 3-89 incorporates autosizing attributes in such a combination that the window can shrink to a very small size and still be usable.

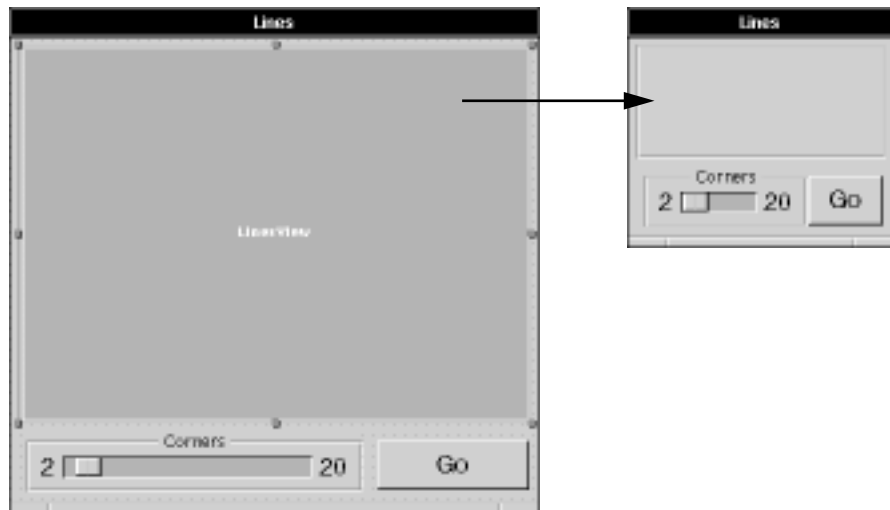


Figure 3-89 Original and Resized Windows

The window's minimum size is set to a dimension just large enough for the main view to show content and for the slider and button to be manipulated (see Figure 3-90).

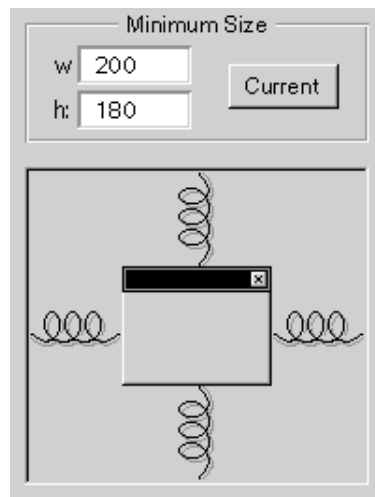


Figure 3-90 Minimum Size Set for Window

The resizing behavior set for the box containing the slider (see Figure 3-91) ensures that the box keeps the same distance from the window's adjacent edges, but resizes the gaps between itself and the other views. It resizes itself horizontally, but not vertically.

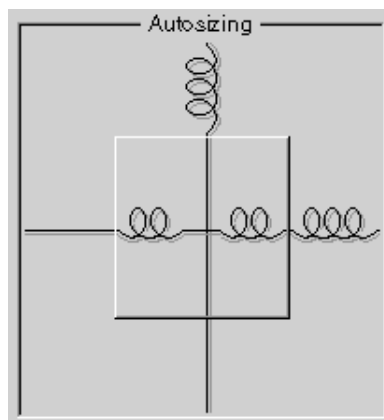


Figure 3-91 Autosizing Behavior Set for Box

The button's autosizing attributes (see Figure 3-92) complement the box's attributes. It keeps the same distance from the window's adjacent edges, but resizes all other distances. It also resizes itself horizontally, but not vertically.

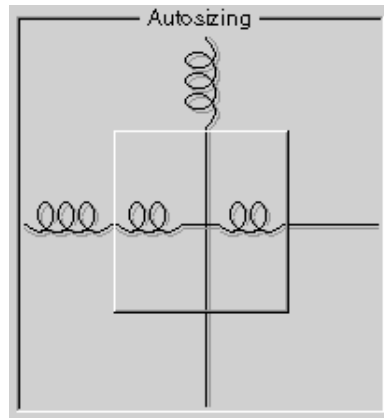


Figure 3-92 Autosizing Behavior Set for Button

The main view of the interface (a custom view) maintains a constant distance from the window's edges, but is itself resizable in all directions (see Figure 3-93).

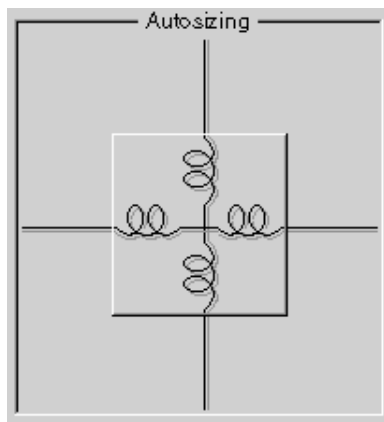


Figure 3-93 Autosizing Behavior Set for Custom View

Using Tags

Tags are integers that you use in your code to identify objects. They offer a convenient alternative to such methods of object identification as fetching an object's title. (What if the object's title changes while the application is running, or the object has no title?) Tag integers can also carry useful information associated with an object, and thus make it easier to integrate that information into a program. Tags are commonly assigned to matrices and to the cells contained by matrices.

You can specify tag integers in the Tag fields of most Attributes displays, as shown in Figure 3-94. To use tag integers for objects in your interface, do the following:

- 1. In Interface Builder, specify the tag integers for objects.**
- 2. If the integers are not intrinsically meaningful, define constants for them in your source code.**
- 3. Send the tag message to a tagged object to get the integer.**
- 4. Evaluate the integer and act upon it.**

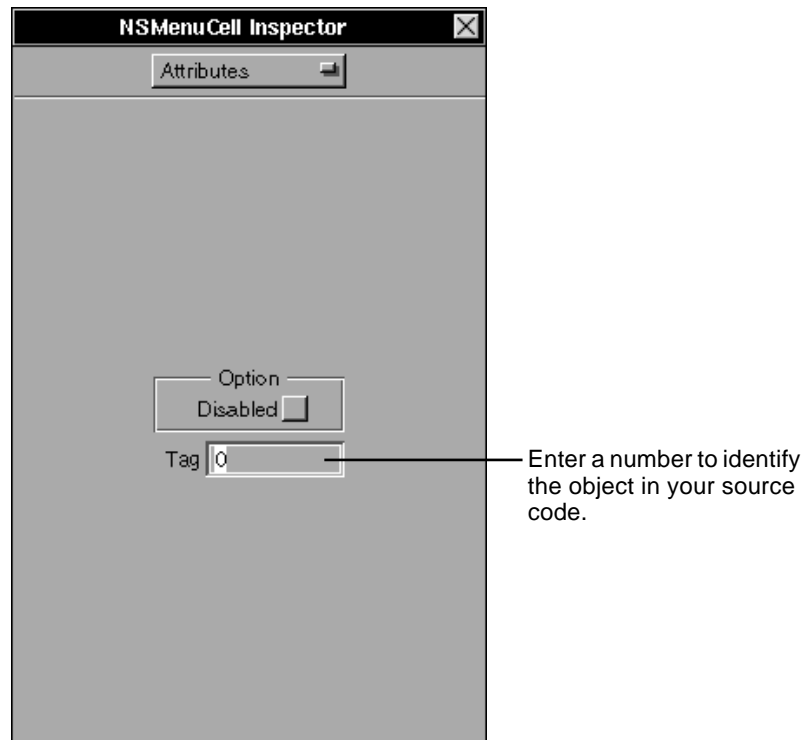


Figure 3-94 Specifying a Tag Integer for an Object

You can also set tag integers programmatically in most `NSView` objects by sending those objects the `setTag:` message.

The integers that you assign could have some intrinsic value; for instance, they could be numbers that are multiplication factors for a document-zoom feature, or numbers that correspond to the number of a keypad in a calculator application. If the tag numbers are not intrinsically meaningful (that is, they are arbitrary), it is prudent to define constants to express them.

```
typedef enum {
    LEFT = 1,
    RIGHT,
    BOTTOM,
    TOP,
    HORIZONTAL_CENTERS,
    VERTICAL_CENTERS,
    BASELINES
} AlignmentType;
```

When you need to identify a tagged objects in your code, use the tag method.

```
- align:sender
{
    [self alignBy:(AlignmentType)[[sender selectedCell] tag]];
    return self;
}
```

Making and Managing Connections

Once you build an interface with objects, you connect those objects so they can communicate with each other. You make connections between objects in Interface Builder by Control-dragging a line between them and then selecting a name for the connection.

Communicating With Other Objects: Outlets and Actions

Outlets

An outlet is an instance variable that points to another object. Objects use outlets to communicate with other objects; they simply send messages to the object identified by the outlet.

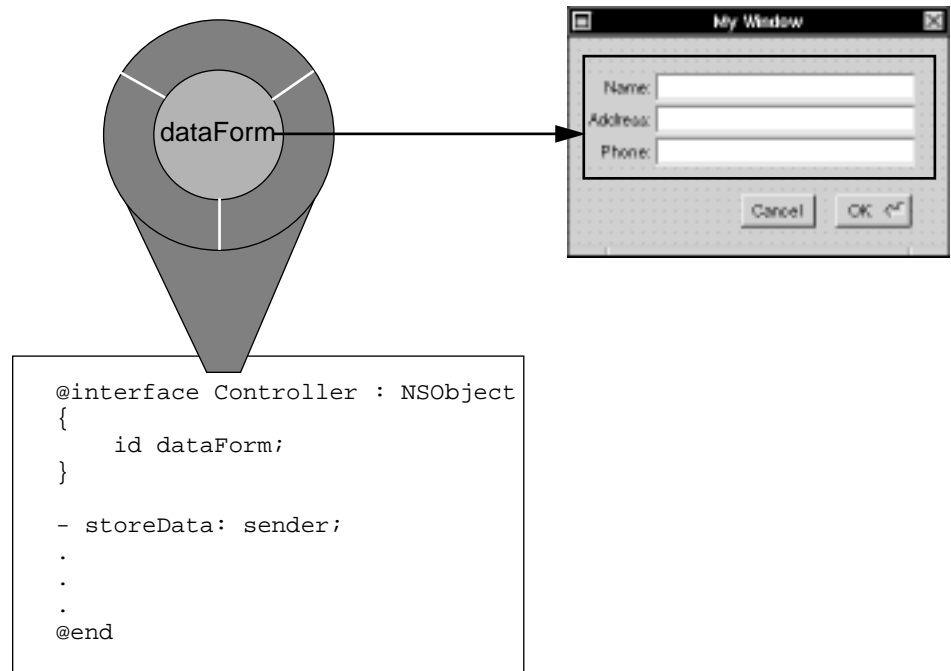


Figure 3-95 Outlet

Using Interface Builder, you can declare and set outlets for the custom objects in your application. You can also set ready-made outlets in many Application Kit objects, such as browsers. Once initialized, the connection information for the outlet is stored in the nib file. At run time, the nib file is unarchived and the outlet is reinitialized with the connection information.

The Application Kit defines two types of outlets that you can use to establish specialized connections with other objects: delegates and targets.

Delegates

A delegate is an object that acts on behalf of another object. Many kit classes define delegate outlets as an alternative to subclassing. All your object must do is register itself as a delegate of the kit object. At important junctures in its life

cycle, the kit object sends messages to its delegate, giving it an opportunity to participate in processing and sometimes even the chance to veto some behavior.

As examples, browsers request their delegates to supply the cells for browser columns; applications inform their delegates when they (the application) are initialized, hidden, and activated.

Targets

Targets are a special kind of outlet. They identify objects that can respond to action messages. When a user activates an `NSControl` object (for instance, clicking a button or moving a slider), that object sends an action message to the target. The action message gives application-specific meaning to the original mouse or key event.

Like a delegate, a target must implement methods to respond to the messages it is sent. But unlike a delegate, which receives messages chosen from a limited set defined by a kit, a target responds to action messages defined by the programmer.

You can also make one object a target of a second object programmatically by sending that second object `setTarget:`.

Actions

`NSControl` objects translate the event messages they receive when users manipulate them into messages meaningful within the application. They then send these messages to other objects. These application-specific messages initiated by an `NSControl` object are called action messages, and the method they invoke are called action methods. An `NSControl` object is simply a user-interface device that permits the user to give instructions to the application, a device that mediates between the user and the object that will ultimately respond to the user's event.

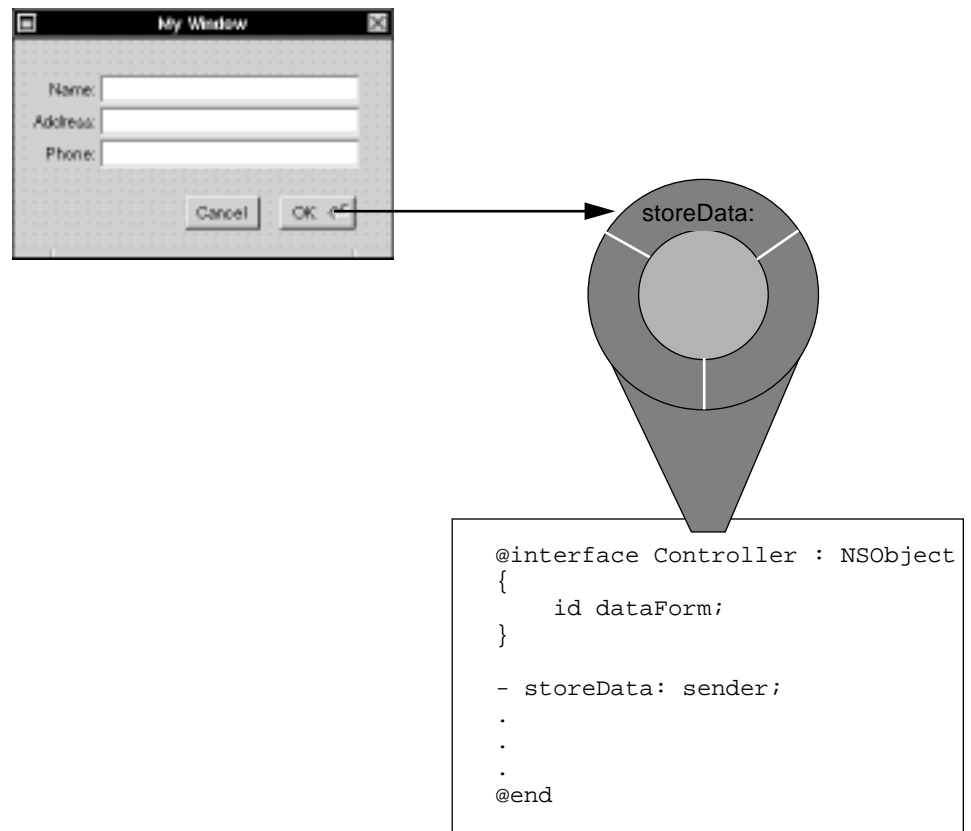


Figure 3-96 Action

NSControl, an abstract class, defines for its many subclasses (such as NSButton, NSScroller, NSTextField, and NSForm) a paradigm for inter-object communication—action messages. But NSControl objects do not act alone: they always contain one or more objects of NSActionCell or its subclasses. The NSActionCell superclass defines instance variables for the two elements essential to an action message:

- Target: the object that is responsible for responding to the user's action on the NSControl
- Action: the method that specifies what the target is to do

Action methods take a single argument, the id of the `NSControl` object that sends the message. This argument enables the receiver to ask the control for more information, if it's needed.

An `NSControl` can send a different action message to a different target for each `NSActionCell` it contains. `NSControls` dispatch action messages differently; for instance, an `NSButton` generally sends action messages on a mouse-up event, but an `NSSlider` usually sends action messages continuously, as long as the mouse button is pressed.

Connecting Objects

In an object-oriented application, isolated objects have little value; they need to send messages to each other to get the work of the application done. Interface Builder gives you a way to establish connections between objects.

To connect two objects, do the following:

- 1. Select an object.**
- 2. Control-drag a connection to another object.**
- 3. In the Inspector panel's Connections display, select an outlet or action.**
- 4. Click the Connect button.**

Begin making a connection in Interface Builder by Control-dragging a connection line from one object to another object. Almost any object will do. Usually you Control-drag a line between an object in the interface and an object in the Instances display.

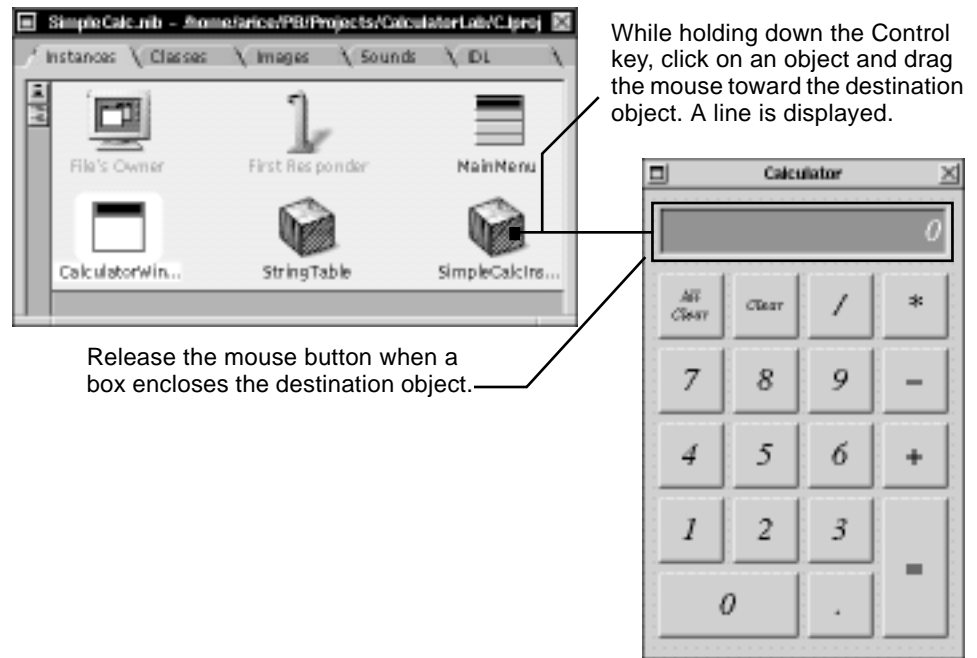


Figure 3-97 Connecting Two Objects

When you release the mouse button, the Inspector panel becomes the key window (see Figure 3-98 on page 3-115). Its Connections display shows the current and potential connections for the destination object.

Outlet Connections

In the example in Figure 3-97, the connection is made from a controller object—a custom object that manages the application—to a text field. The controller object (`SimpleCalcInstance`) declares several outlets—identifiers of destination objects—as instance variables.

When you make a connection between objects, the first column of the Connections display shows the source object's outlets ("source" meaning the object from which a connection line is drawn).

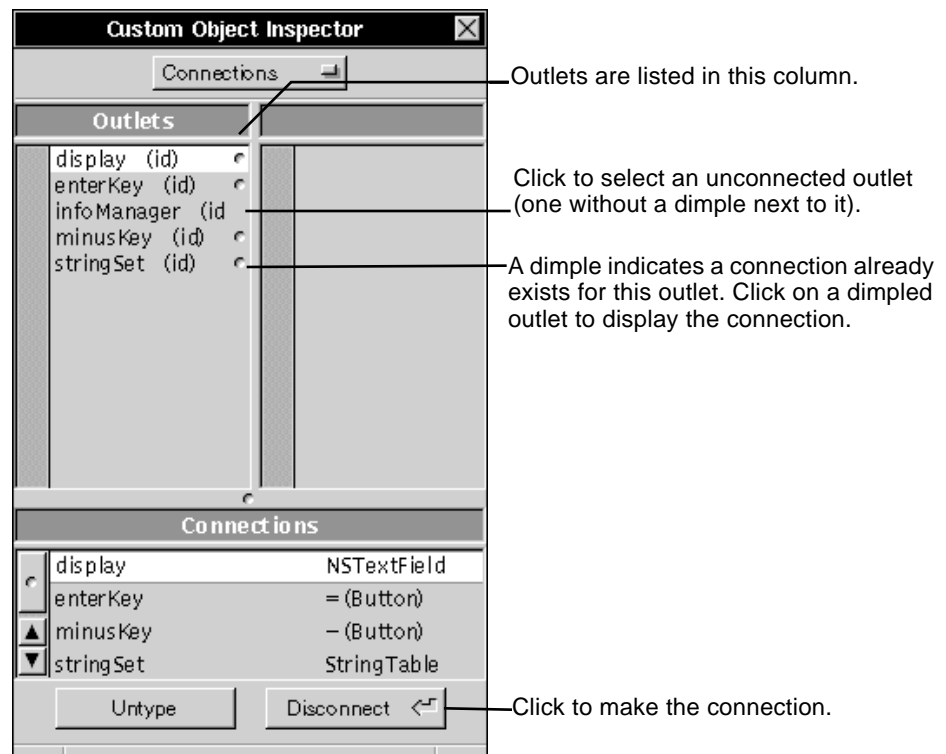


Figure 3-98 Inspecting an Outlet Connection

You can make outlet connections between objects in the Instances display, as shown in Figure 3-99 on page 3-116.

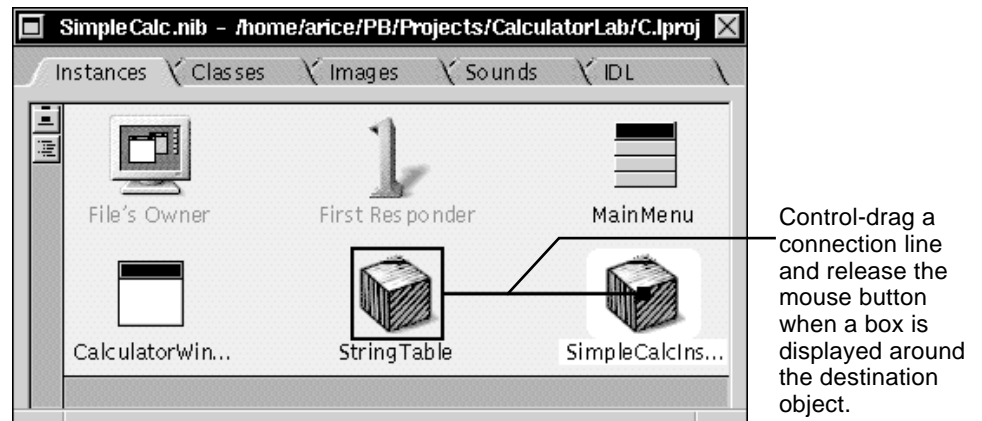


Figure 3-99 Connecting Objects in the Instances Display

Action Connections

When you make a connection by dragging a line from an `NSControl` object in the interface—a button, slider, text field, menu command, pop-up list, or matrix—odds are that the destination object is a target, and that you can complete the connection by selecting an action method, as shown in Figure 3-100 on page 3-117.

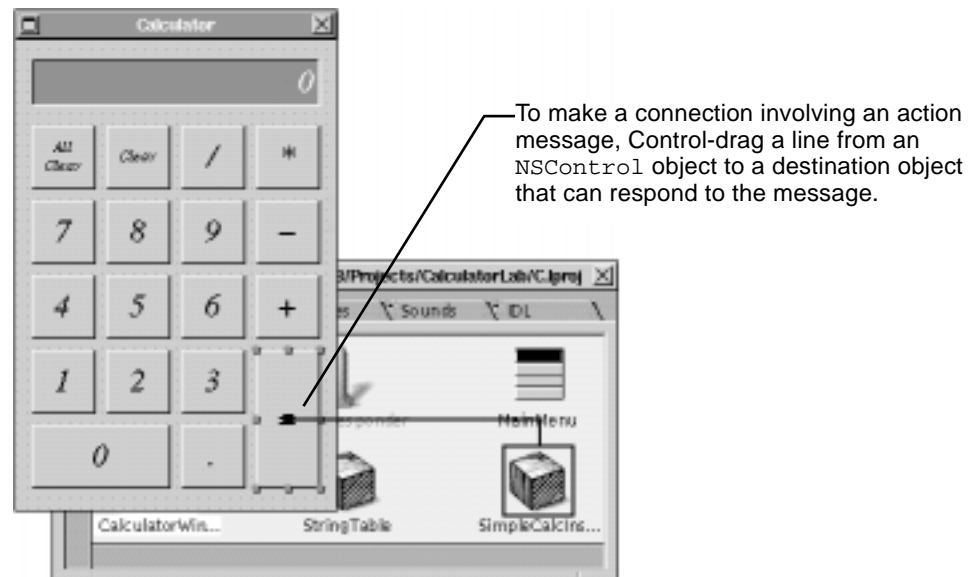


Figure 3-100 Making an Action Connection

The destination object in an action connection is frequently a custom object that manages the application or a particular window (controller object).

When you make a connection from an `NSControl` object, the Inspector panel becomes key and shows the Connections display for the destination object.

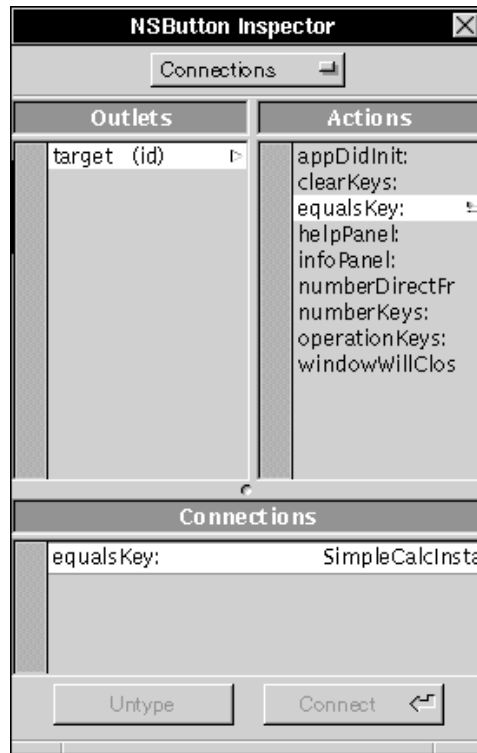


Figure 3-101 Inspecting an Action Connection

When the user manipulates the `NSControl` object, such as clicking on a button or moving a slider, the action message is sent to the destination object (the target).

Connections Within the Interface

Sometimes you can connect two objects on an interface. These connections can involve both outlets and actions. Often one of the objects is a custom `NSView` object, as in Figure 3-102 on page 3-119.

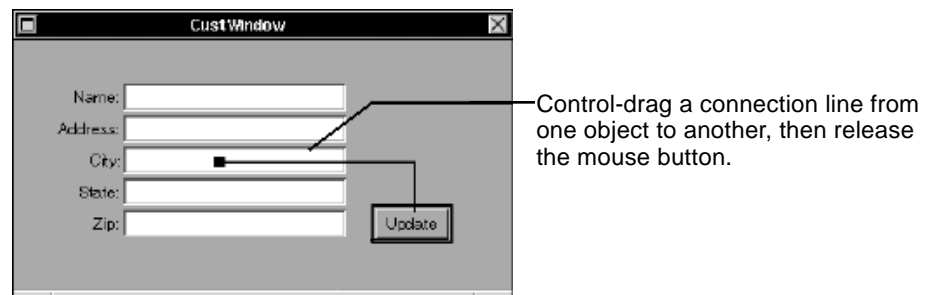


Figure 3-102 Connecting Objects within an Interface

Connections within an interface can also involve two Application Kit objects. Two examples are interconnecting text fields (so the user can tab from field to field), and connecting a menu command such as Print to an `NSText` object.

Note – To enable printing of an `NSText` object, drag a connection line from the Print menu command (or other `NSControl` object that initiates printing) and select the `printPSCode:` action in the Connections display.

Outlets are destination objects specified as instance variables. Actions are methods that `NSControl` objects (such as buttons) invoke in another object. See “Communicating With Other Objects: Outlets and Actions” on page 3-109 for more information.

“Creating a Class” on page 3-137 describes connecting the outlets and actions of custom objects in the context of creating a class.

See “Communicating With Other Objects: Outlets and Actions” on page 3-109 for more information on targets and actions.

See “Compound Objects” on page 3-91 for descriptions of the interaction between `NSControl` objects and `NSCell` objects, and of the role `NSMatrix` objects play.

You can connect text fields and form fields so that when the user presses the Tab key, the pointer moves to another field. See “Enabling Inter-Field Tabbing” on page 3-128 for information on this procedure.

Making Connections in Outline Mode

You can make connections between objects in the outline mode of the Instances display as well as its icon mode. The connections can be between an object in the outline and an object in the interface (see Figure 3-103) or between two objects listed in the outline (see Figure 3-105 on page 3-122). To connect objects in outline mode, do the following:

1. **Select an object.**
2. **Control-drag a connection to another object.**
3. **Specify an outlet or action in the Connections display for the destination object.**

Before you make a connection involving an object in outline mode, make sure that the object is visible in the display. (You might have to expand the object's "parents" in outline mode to do this.)

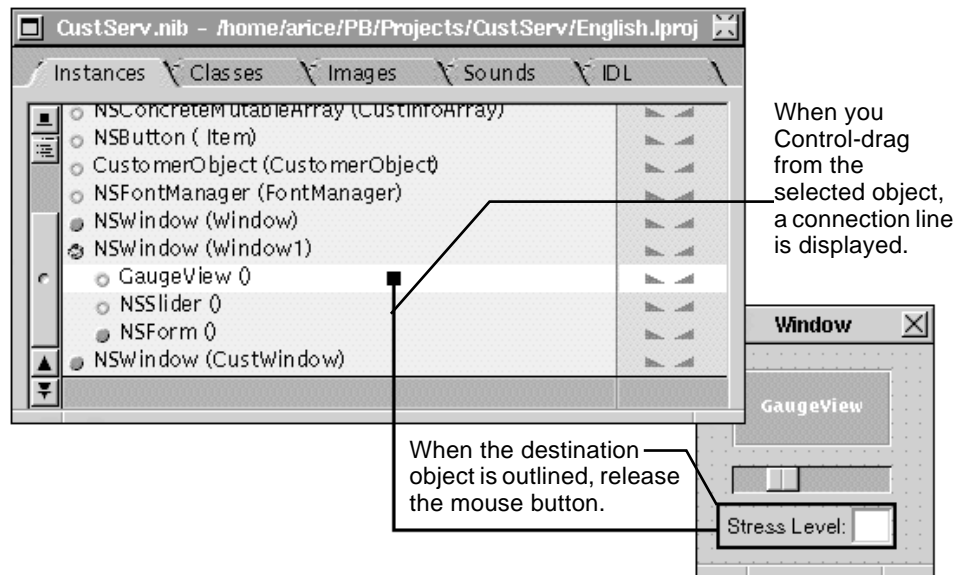


Figure 3-103 Connecting an Object in the Outline with an Object in the Interface

The Connections display of the destination object's Inspector lists the possible connections, as shown in Figure 3-104 on page 3-121.

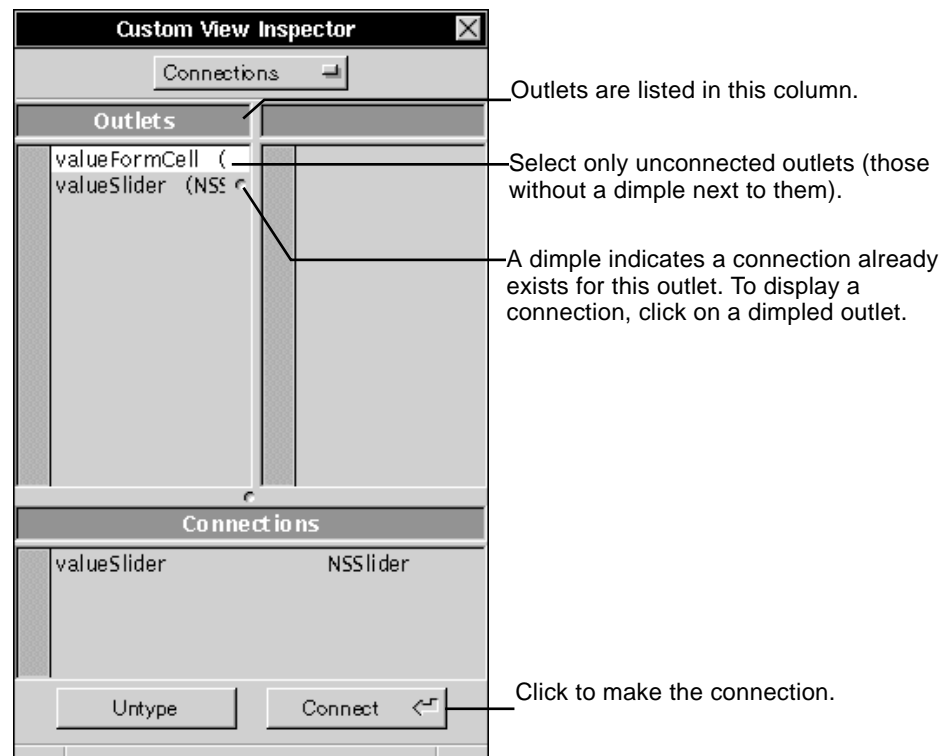


Figure 3-104 Displaying the Possible Connections

The outline mode offers the useful capability for making connections without leaving the nib file window. In this example, the same connection is made as in Figure 3-103 on page 3-120.

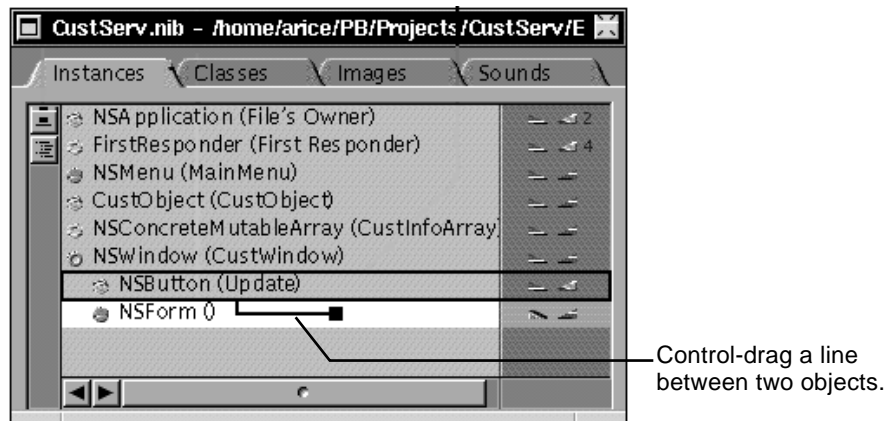


Figure 3-105 Making a Connection within the Nib FileWindow

When the destination object is outlined, its Connections display lists the possible connections. Complete the connection as described above.

Examining Connections

- In the interface: Select an object and look at the Connections display of the Inspector panel.
- In the Instances display: Select an object and look at the Connections display of the Inspector panel.
- In the Connections display: Click a dimpled outlet to see the connection line drawn.
- In outline mode: Click a triangle button in the column to the right of an object.

Interface Builder gives you many ways to examine and verify connections between objects. It makes it easy, for example, to discover what outlets and actions might be associated with an object in the interface (see Figure 3-106).

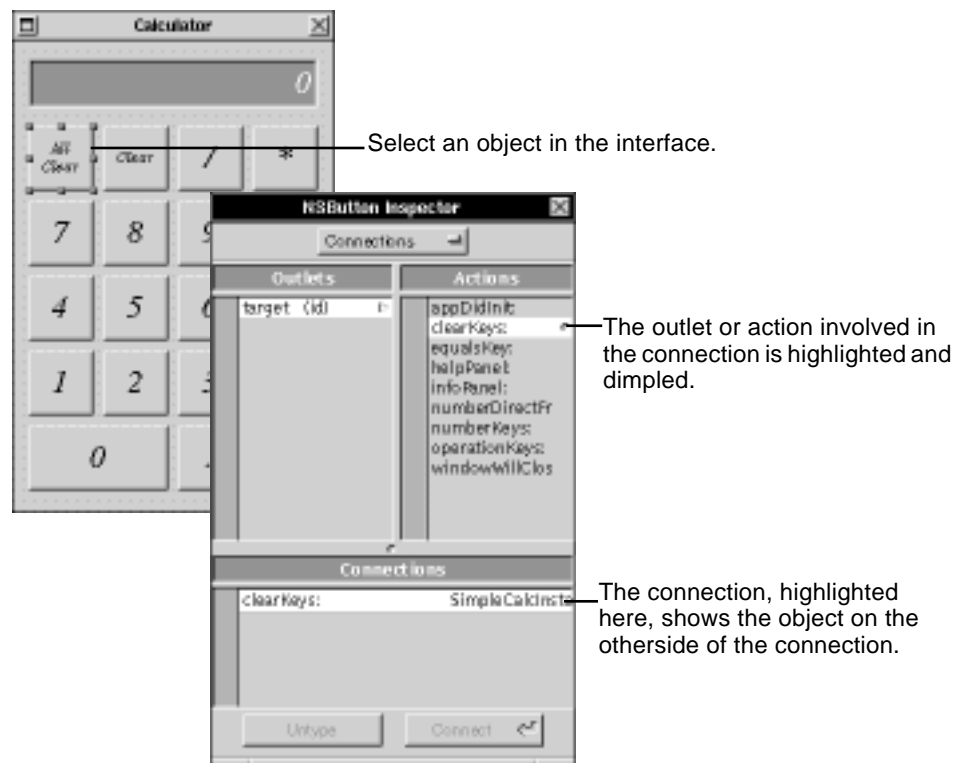


Figure 3-106 Displaying the Outlets and Actions Associated with an Interface Object

You can also select an object in the Instances display (in both icon and outline modes) and examine the Inspector panel as described above to find out what object it is connected to.

You can also examine object connections going in the other direction, from the Connections display to the interface and the Instances display (see Figure 3-107 on page 3-124).

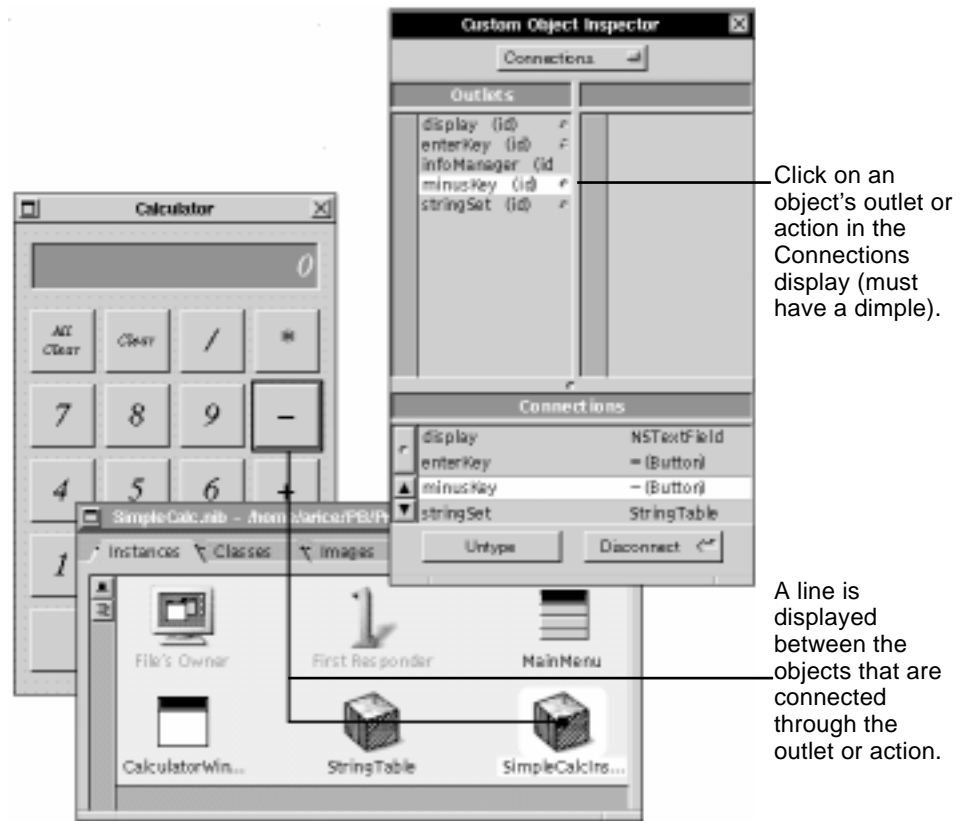


Figure 3-107 Examining a Connection through the Inspector Panel Connections Display

The Connections display allows you to see one connection at a time. The outline mode of the Instances display (see Figure 3-108 on page 3-125) shows you all connections an object has, both connections into the object and connections from that object to other objects.

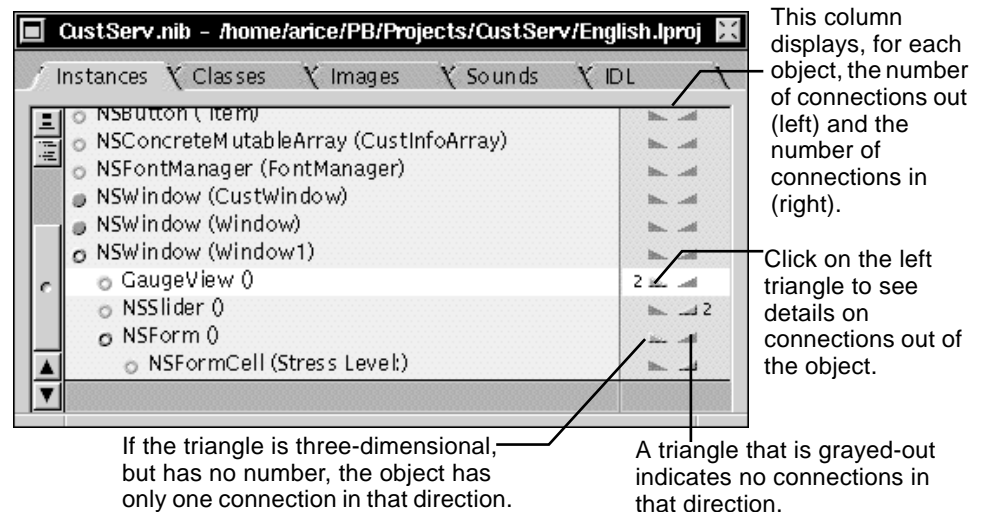


Figure 3-108 Checking Connections in the Instances Display

When you click a three-dimensional triangle, lines appear to show the connections between objects, as shown in Figure 3-109 on page 3-126. The name and class of each connected object is highlighted in bold. Each connection is labelled with the name of an outlet or action.

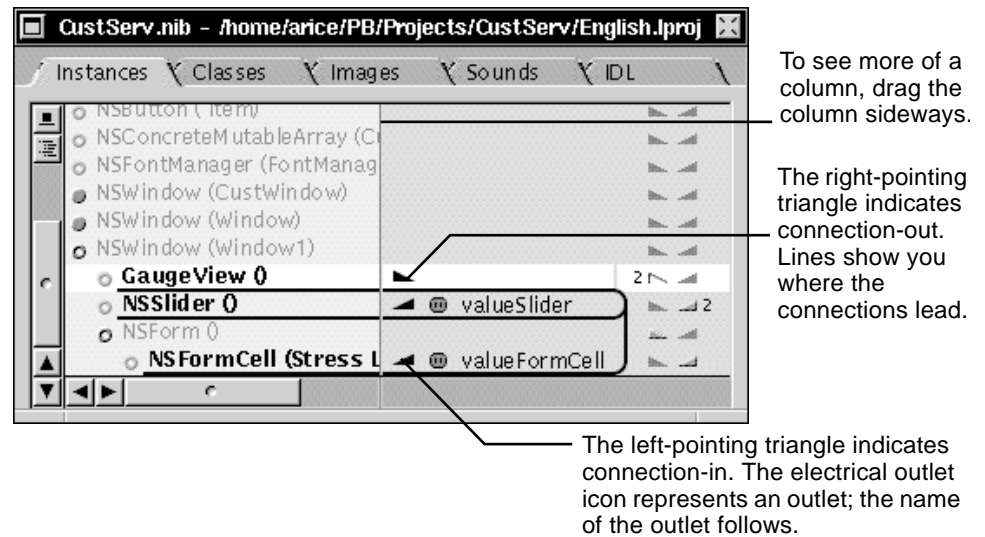


Figure 3-109 Looking at Connections Out in the Instances Display

To see connections into an object, as shown in Figure 3-110, click on a three-dimensional triangle that points to the left (that is, a triangle on the right side of the connections column).

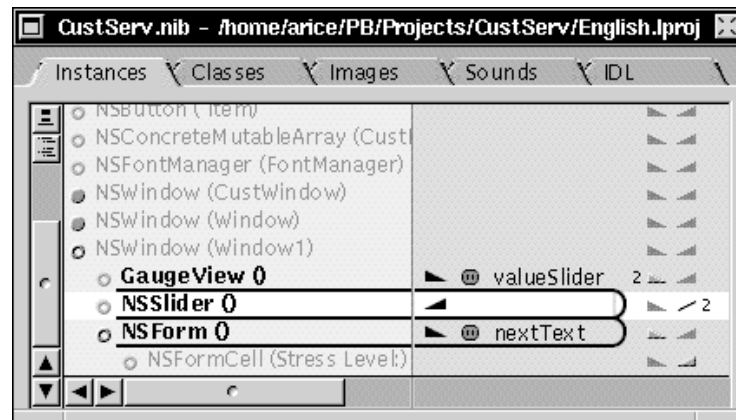


Figure 3-110 Looking at Connections In in the Instances Display

An object may have multiple connections with another object, both in and out, both outlets and actions. In these cases, the outline mode lets you toggle between the connections.

To make the connection lines disappear, click the three-dimensional triangle button that is highlighted.

Identifying Objects in Outline Mode

- To see a representation of an object, Alt-click on it in outline mode of the Instances display.
- To have an arrow point at the interface object, Control-Shift-click on the object in outline mode.

In the outline mode of the Instances display you might want to verify what an object is before connecting it to another object. You have two graphical ways to identify an interface object. One method displays an image representing a selected object, as shown in Figure 3-111.

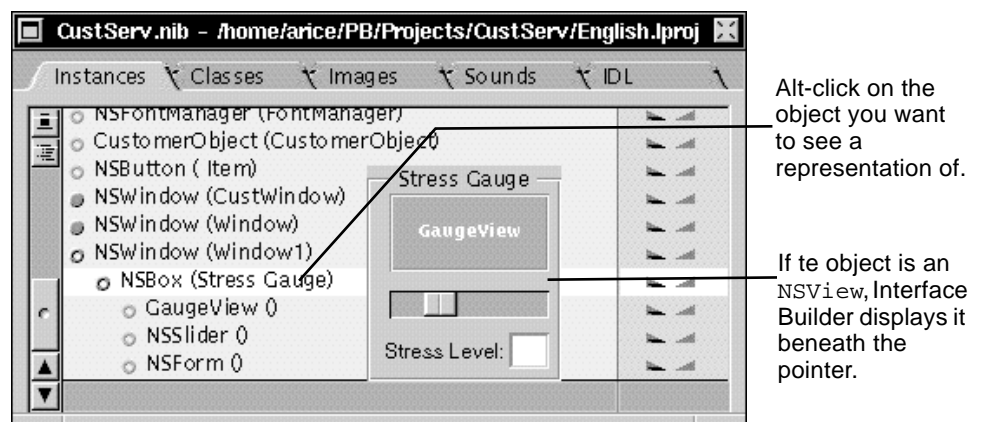


Figure 3-111 Displaying an Image Representing the Object Selected in the Outline

When you Alt-click on non-NSView objects in outline mode, the images that represent them in icon mode are displayed (cubes for custom objects, mini-windows for panels and windows). Menu, First Responder, and File's Owner do not display icons.

The other technique locates an object in the interface with a large arrow, as shown in Figure 3-112.

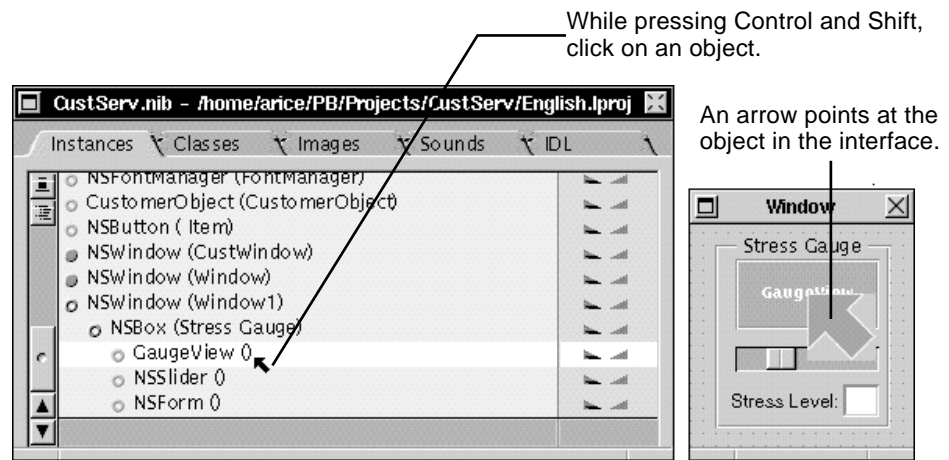


Figure 3-112 Locating the Object in the Interface with an Arrow

Control-Shift-Clicking on Menu, File's Owner, and First Responder has no effect.

See “Outline Mode of Instances Display” on page 3-12 for an introduction to outline mode.

Enabling Inter-Field Tabbing

Sometimes when users press the Tab or Return key in a window with multiple fields, you want the pointer to jump from the current text or form field to the next field. When users press Shift-Tab, you want the pointer to go the previous field. An NSForm object (a matrix) automatically moves the pointer between its fields. But between text fields, between NSForms, or between an NSForm and a text field, you must specify this behavior.

To specifying inter-field tabbing, do the following:

1. **Make a connection line between forms or fields.**
2. **Select nextText in the object's Connections display.**
3. **Click the Connect button.**

The `NSMatrix` class (of which `NSForm` is a subclass) and the `NSTextField` class define an instance variable, `nextText`, as an outlet. This is what you connect (see Figure 3-113).

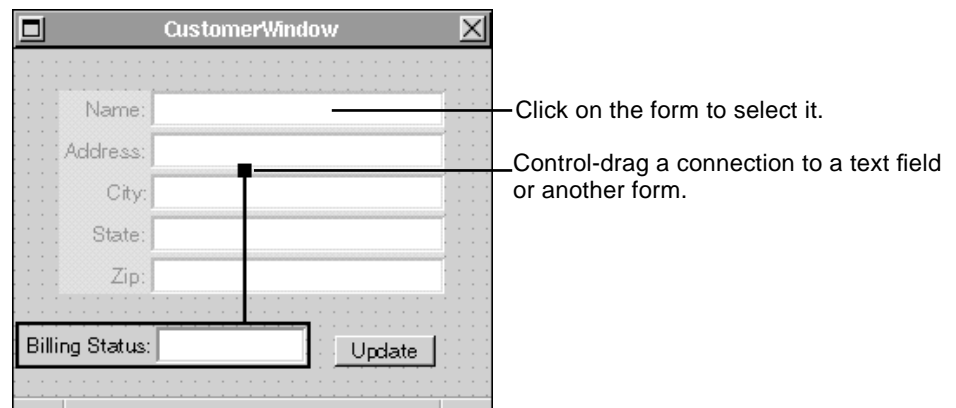


Figure 3-113 Connecting Two NSForm Objects

Next, make the connection in the Inspector panel as shown in Figure 3-114 on page 3-130.

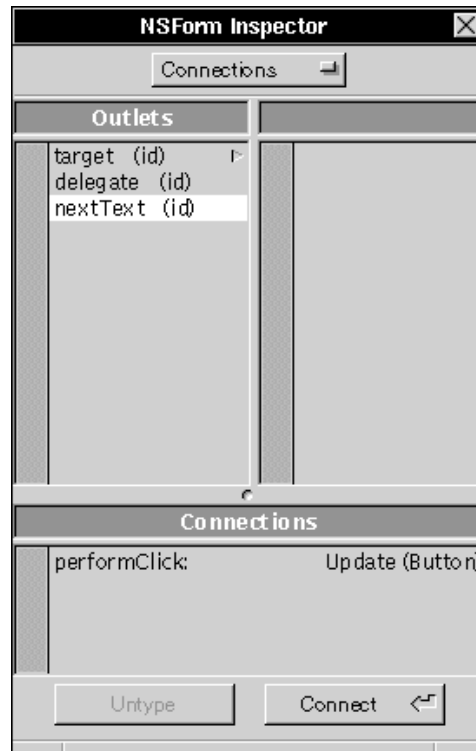


Figure 3-114 Making the Connection in the Inspector Panel

Note the `textDelegate` outlet in this example's Inspector. This is the object that receives delegation messages from the `NSText` class on behalf of a text-editable field.

Disconnecting Objects

Interface Builder gives you two ways to break the connections between objects. The first method uses the Inspector panel, as shown in Figure 3-115 on page 3-131, through the following steps.

1. **Select an object.**
2. **In the Connections display, select a connection.**

3. Click Disconnect.

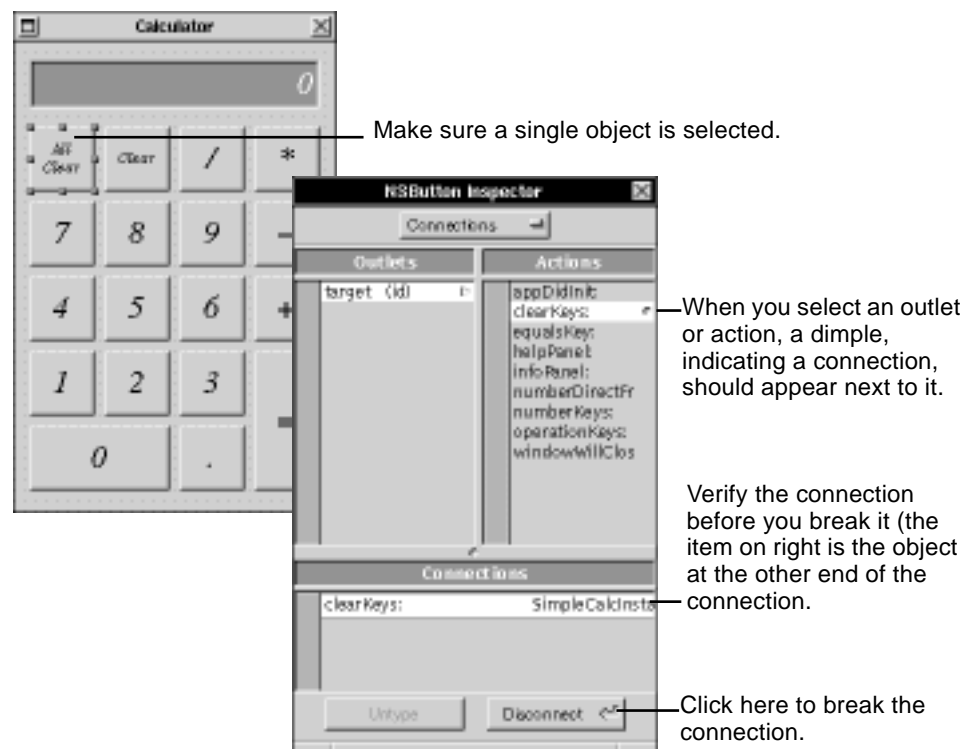


Figure 3-115 Disconnecting Objects Using the Inspector Panel

You can also initiate this procedure by selecting objects in icon mode of the Instances display, and then disconnecting them in the Inspector panel as above.

The alternative method for disconnecting objects is somewhat easier because you can complete the operation in one place: in outline mode of the Instances display. First show connections for an object by clicking a three-dimensional triangle button, as shown in Figure 3-116 on page 3-132.

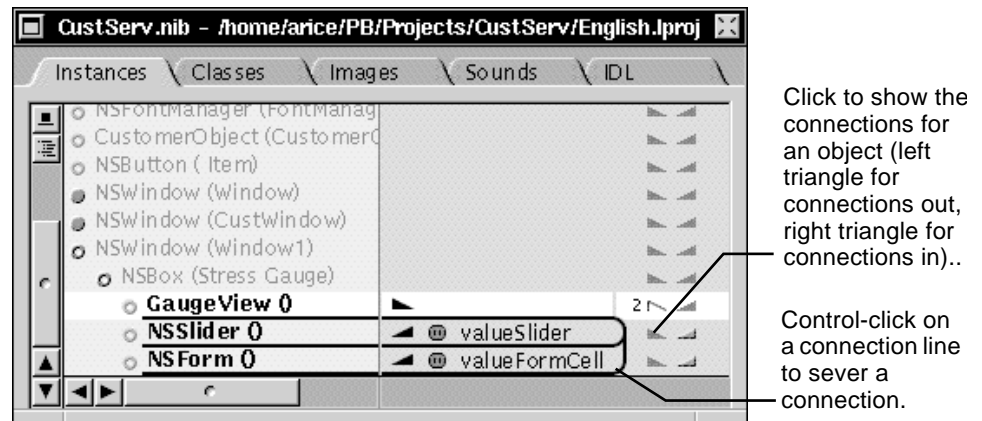


Figure 3-116 Disconnecting Object in the Instances Display

To make the scissors pointer appear over a connection line, you must press the Control key over a line on the right side of the column divider (nearest the connection-out and connection-in triangle buttons). You Control-click on the left side of the column divider to begin connection operations. See “Examining Connections” on page 3-122 to learn how to use outline mode to display the connections between objects.

Attaching Help to Objects

The Help Builder panel makes it easy to associate help text with any object in your application’s user interface. (To learn about the design of the OpenStep help system, see the `NSHelpPanel` class specification in *OpenStep Programming Reference*.)

The Help Builder panel is a slightly modified version of the standard Help panel.

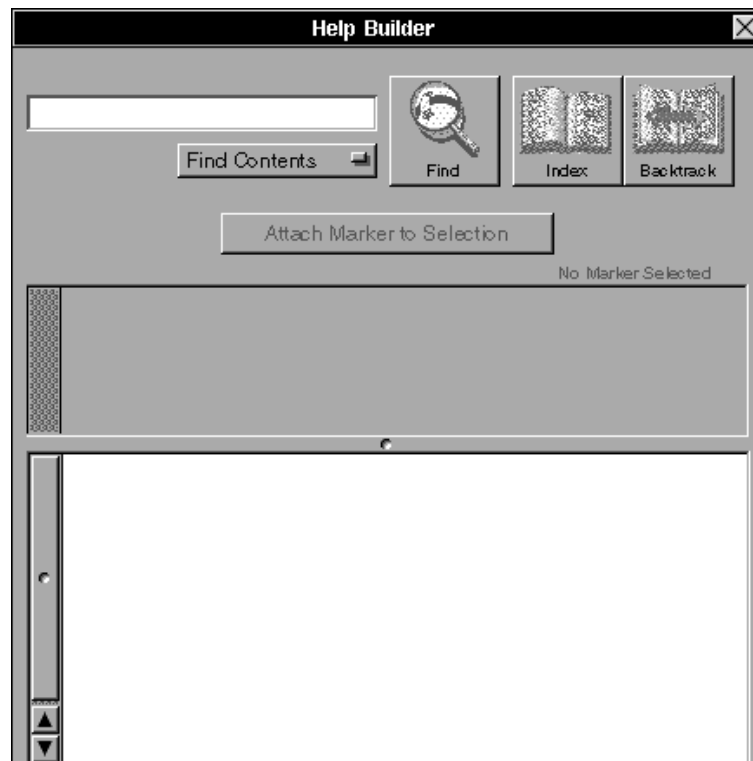


Figure 3-117 Help Builder Panel

Attaching help to an object involves selecting an object in your application, displaying the help text in the Help Builder panel, optionally selecting a help marker within the text, and clicking on the AttachFile to Selection button. Thereafter, when the application runs and the user Help-clicks on the object (that is, holds down the Help key and clicks on the object), the specified help text is displayed in the application's Help panel. However, before you begin attaching help text to your application's objects, you must provide your application with two components: a Help menu item and a `Help` directory.

Interface Builder's Menu palette supplies an Info menu item that, when dragged to your application's main menu, reveals a submenu containing a Help menu item. This menu item is preconfigured to open the Help panel. (If you inspect the Help item's connections, you will see that it sends a `showHelpPanel:` message to the First Responder object.)

Project Builder can provide your application with the required Help directory. Choose the Add Help Directory command in Project Builder's Project menu to create this directory. Project Builder creates the directory within the `.lproj` directory of your chosen development language (for example, `English.lproj/Help`). It copies into this directory generic table-of-contents and index files.

The next step is to customize these files and to add content files of your own. The generic help text that is accessed through the supplied table-of-contents and index files gives help on basic operations, such as using the mouse and choosing commands. You will want to add files that describe the operations that are unique to your application. You can also override or eliminate any of the generic help text that is not applicable to your application.

You create help files using Edit. (Make sure that Edit is in Developer Mode so that the Help commands can be accessed from the Format menu.) Perhaps the easiest way to ensure that the files you add agree in style and formatting with the generic help files is to display a generic file, copy its contents, and paste it into a new Edit document. Be sure to resize the new document's window to the same width as the original so that the text wraps to the same margins. You can then modify the contents of the new help document and save it in the `Help` directory. If you think you will want to associate objects with specific passages within the file, rather than to the file in general, you can place help markers within the document.

Each file you add should be represented by a new entry in the table-of-contents file. (However, see the `NSHelpPanel` class specification for an exception to this rule.) After adding content files, you will also probably have to update the index.

Once the table-of-contents, content, and index files for your help system are finished, you can begin attaching help to your application's user-interface objects. Display the Help Builder panel by choosing the Help Builder command from Interface Builder's Tools menu or by clicking on the Help Builder button in the Help display of the Inspector panel. Select an object in your application's user interface, locate the relevant help text in the Help Builder panel, and click on the Attach... button. If the Help inspector is open, it displays this new association in its Help Attachments list.

The Help Builder panel offers several ways to locate specific portions of help text. First, you can use the table-of-contents or index displays to locate a file. In addition, the pop-up list below the Find field lets you search for help files by name, for marker names within the help files, or for any string.

Reviewing Help Attachments

The Help display, shown in Figure 3-118, lets you review attachments between objects in your application and help text. It also gives you access to Interface Builder's Help Builder panel.

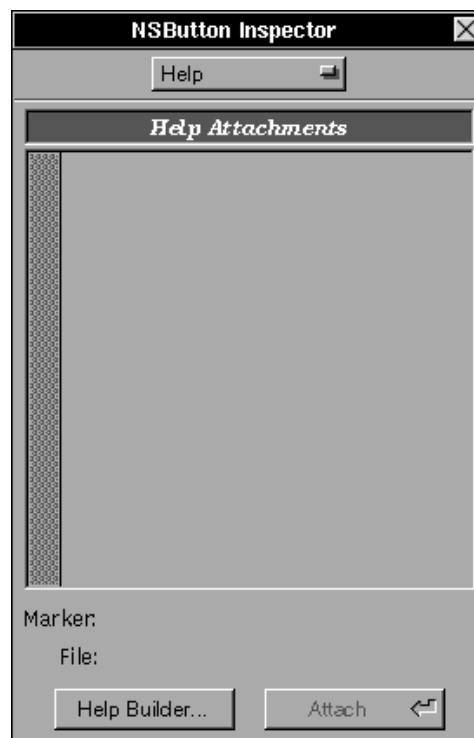


Figure 3-118 Help Display

The Help display is used in conjunction with the Help Builder panel. See “Attaching Help to Objects” on page 3-132 for information on associating help text with objects in your application.

Assuming you have attached help to objects in your application, the Help display of the Inspector panel will list those attachments. Each entry in the list has two parts. The left half of the entry identifies the object, and the right half displays the file name for the attached help. Below the Help Attachments list are two text fields. The Marker field names the marker that the object is attached to within the help file. If the object is not attached to any marker in the file, the Marker field is blank. The File field displays the path of the help file relative to the application's `Help` directory. If the entire path is not visible, scroll the text field horizontally to reveal the hidden portion.

You can remove an attachment by selecting it in the list and clicking on the Detach button.

Testing the Interface

After you create an interface, Interface Builder lets you see how it works from the user's perspective. Just choose the Test Interface command from the Document menu.

- 1. Choose the Test Interface menu command.**
- 2. Check the functioning of kit objects.**
- 3. Choose Quit from the application menu or double-click the switch icon in the application dock.**

Interface Builder's menu, windows, and panels disappear, leaving only the actual interface and (if you are testing the application's main nib file) the main menu. Give your interface a test ride. Here are some of the things you might try:

- Verify that the pointer moves from field to field when you press Tab and Return.
- Verify that you can copy, cut, and paste text (First Responder actions).
- See if you can print (the Print menu item must be connected to an appropriate `NSView` object's `printPSCode:` action method).

Note – When you test your interface, the behavior provided by your custom classes is not called into play (with the exception of static, compiled palette objects). You can only test the behavior that kit and static palette objects exhibit in themselves and when they send messages to each other. To test all components of your application, you must compile and run it.

When you are finished testing the interface, exit from test mode.

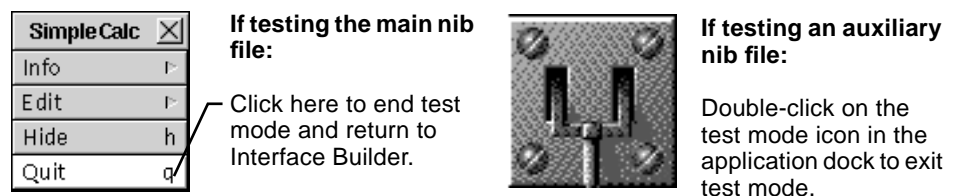


Figure 3-119 Exiting Test Mode


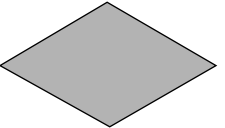


Creating a Class

Creating a class, or adding an existing class, is not a set of discrete, modular tasks, but a process consisting of many interdependent tasks. The order of these tasks is significant; with some exceptions, you need only follow the tasks in the sections that follow sequentially, from first task to last task, and you will create a useful class.

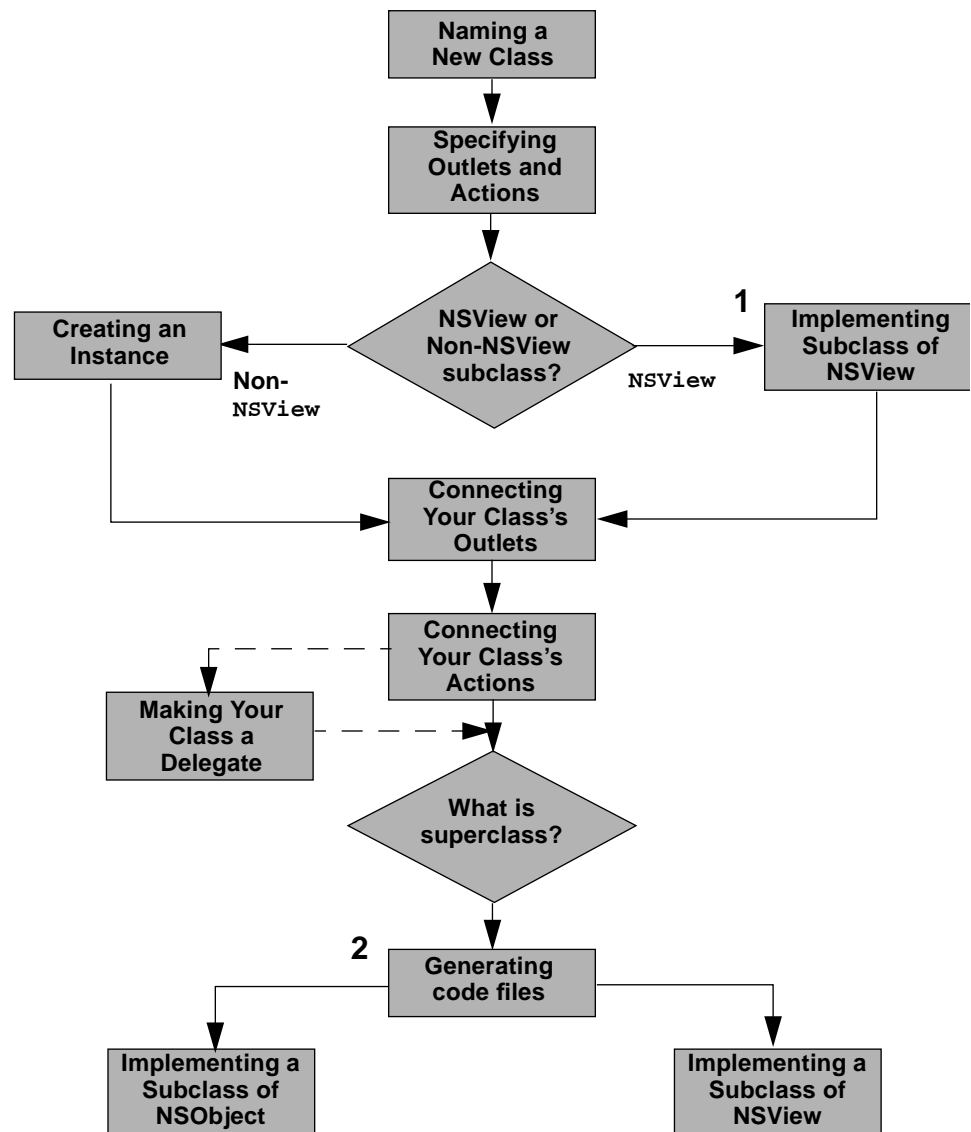
However, the exceptions are significant, so flowcharts are provided to point the way. Figure 3-120 on page 3-139 guides you through the tasks required to define and implement a subclass of a root class or the `NSView` class.

Figure 3-120 on page 3-139 identifies the tasks you must complete to integrate an existing class into an application. Table 3-8 on page 3-138 explains the symbols used in the flow charts.

Table 3-8 Flow Chart Legend

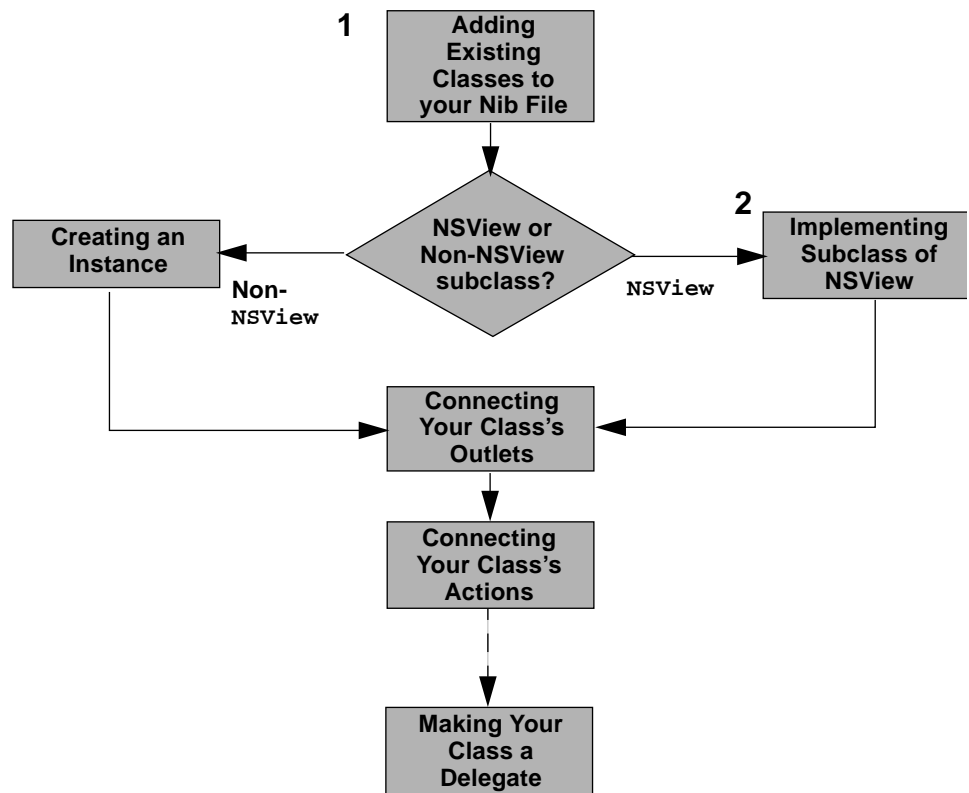
Symbol	Meaning
	Main Flow
	Decision Point
	Optional Flow
	Section in this chapter that describes a task you perform

Note – Interface Builder’s role in subclass creation is to help you locate the class in the hierarchy, name it, connect an instance of it with other objects in an application, and generate template source files. When Interface Builder’s role is finished, you then must make the most important contribution: the source code that gives your class its distinctive behavior.



1. If you branch to "Implementing a Subclass of NSView" after specifying outlets and actions, complete only the step "Making an Instance of an NSView Subclass" in this task for now, and go on to the next task
2. After generating code files, you must switch over to Project Builder and open the header (.h) and implementation (.m) files for the class in Edit or some other code editor.

Figure 3-120 Flowchart for Defining and Implementing a Subclass of a Root Class or NSView



1. You will probably want to add your class's header (.h) and implementation (.m) files to Project Builder as well as Interface Builder. See "Adding Files to a Project" on page 2-17. for information on this procedure.
2. If you branch to "Implementing a Subclass of NSView" after specifying outlets and actions, complete only the step "Making an Instance of an NSView Subclass" in this task for now, and go on to the next task

Figure 3-121 Flowchart for Integrating an Existing Class into an Application

Naming a New Class

When you create an application in OpenStep, you must create at least one subclass to do anything meaningful. The Application Kit, Foundation Kit, and other OpenStep kits are powerful frameworks that do much of the work for you, but you must always supply one or more subclasses, the distinctive and logical flow of your application. To create a subclass, do the following:

1. Choose the Classes display of the nib file window.
2. Select the class from which you want your subclass to inherit.
3. Choose Subclass from the nib file window's Operations menu.
4. Type the name of your class over the highlighted default name.

When you create a class, the first thing you must do is select your class's superclass. Make your selection in the Classes display of the nib file window, as shown in Figure 3-122.

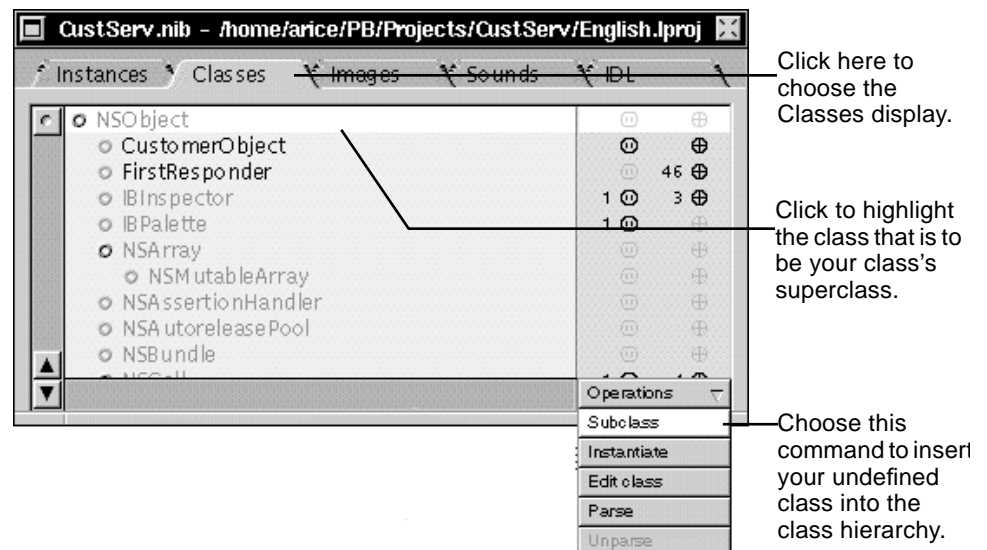


Figure 3-122 Selecting and Subclassing a Superclass

Note – Pressing the Return key when a class is selected is equivalent to choosing the Subclass command.

The new class is listed under its superclass with a default name: the superclass name prefixed with "My", such as `MyNSObject`. Replace this default name with the new name, as shown in Figure 3-123.

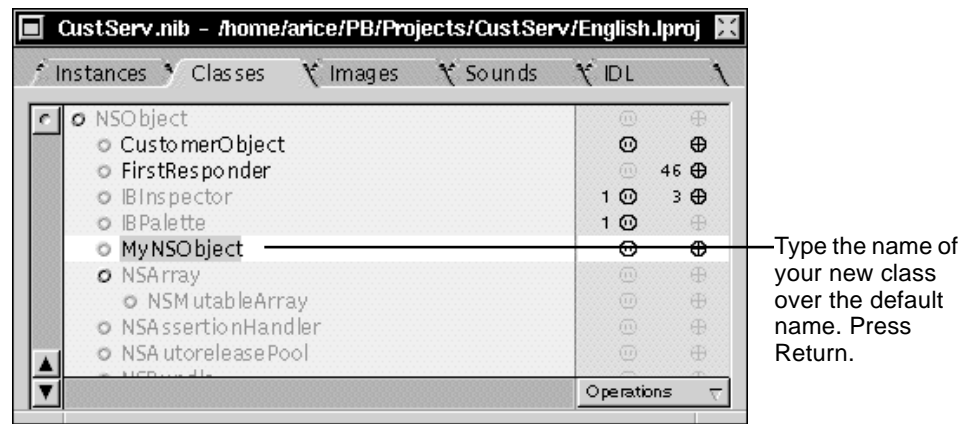


Figure 3-123 Naming the New Class

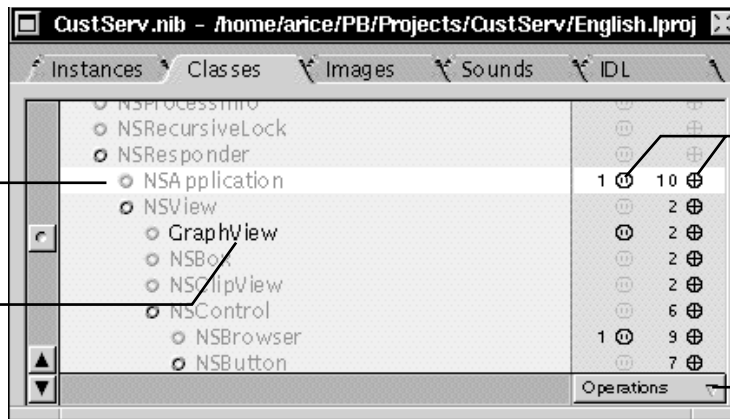
Later, if you want to rename the class, first re-select the class name by double-clicking on it. Then type the new name, replacing the selected text.

A Perspective on Class Hierarchy

The Classes display of the nib file window (see Figure 3-124 on page 3-143) shows the classes of which the current nib file is aware. The display lets you browse through both OpenStep classes and custom classes. It also depicts, by indentation, class inheritance relationships, and reveals the names of each class's outlets and actions.

The Classes display shows hierarchy by indentation; for example, `NSApplication` inherits from `NSResponder`. If the circle button is filled, the class has subclasses that are not shown. Click on the button to display the subclasses.

If the class name is black, it is a custom class. If the class name is gray, it is an OpenStep class.



Click on outlet (electrical outlet icon) and action (cross-hairs icon) buttons to display class outlets and actions.

Use the Operations pull-down list for operations related to creating a class.

Figure 3-124 Classes Display

You can move up and down the list of classes by pressing the up arrow key and down arrow key on your keyboard. When a class is highlighted, you can show its subclasses by pressing the right arrow key, and collapse an indented list by selecting the superclass and pressing the left arrow key. If the nib file window is active, incremental search is active; just type the first few letters of a class name until it is highlighted.

Specifying Outlets and Actions

An object isolated from other objects is of little use. Interface Builder provides two ways for you to specify how objects of your class communicate with the outlets and actions of other objects: outlets and actions. To specify an outlet or action for your class, do the following:

- 1. Click on the outlet button or the action button for the class.**
- 2. Select *Outlets* or *Actions* and press the Return key.**
- 3. Enter the name of the new outlet or action in place of the default name that is displayed.**

Before you begin this task, take a moment to consider what other objects you want instances of your class to send messages to, and what kinds of requests instances of your class are likely to receive from other objects.

Note – For background information on outlets and actions, see “Communicating With Other Objects: Outlets and Actions” on page 3-109.

Adding Outlets

Outlets are instance variables that identify other objects. In the Classes display, you access the outlets of a class by clicking on the electrical outlet buttons, as shown in Figure 3-125.

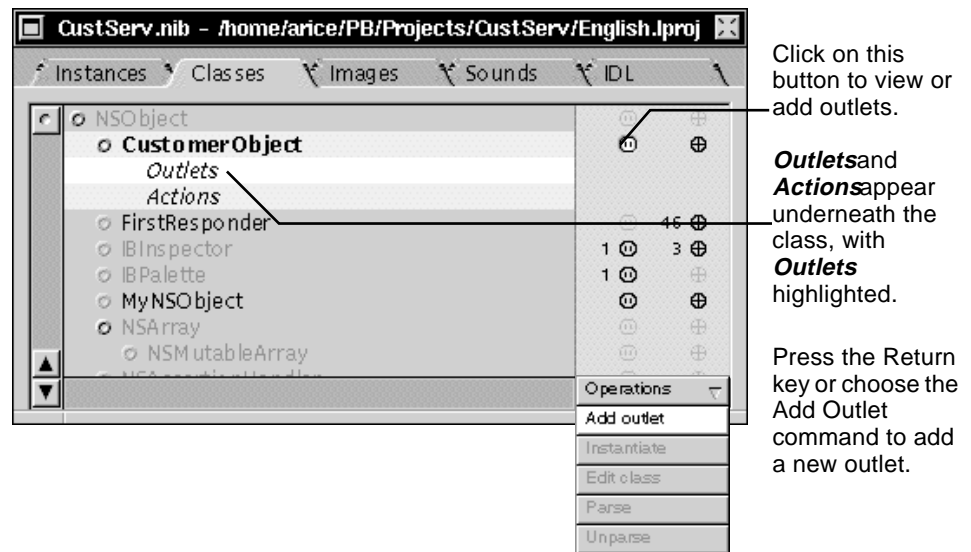


Figure 3-125 Accessing the Outlets of a Class

When you press the Return key or choose the Add outlet command from the Operations menu, a new outlet is displayed under *Outlets*, as shown in Figure 3-126 on page 3-145. Type the name of the outlet in place of the default name and press the Return key.

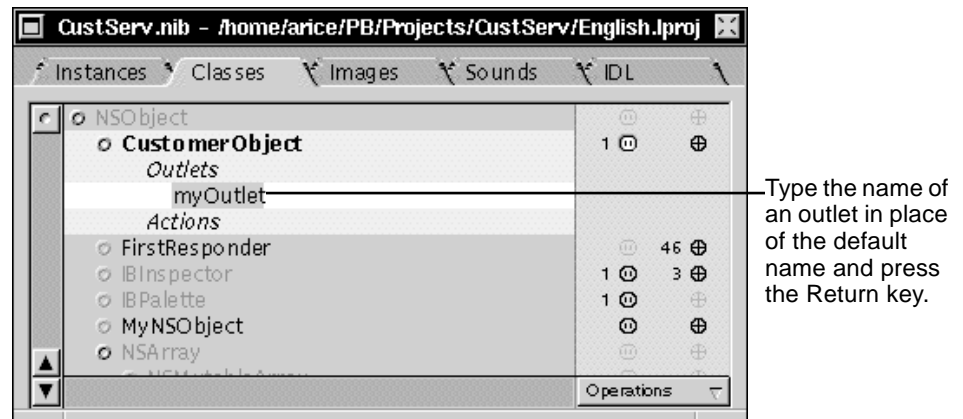


Figure 3-126 Naming a New Outlet

When you press the Return key, the outlet is renamed and Interface Builder highlights the new outlet. To add another outlet, press the Return key again.

Note – To display a class’s outlets and actions (with Outlets highlighted), you can choose the Edit class command from the nib file window’s Operations menu instead of clicking on the outlet button.

Adding Actions

Actions are methods invoked as a direct consequence of the manipulation of `NSControl` objects in your application’s interface, such as when users click on a button. In the Classes display, you access the actions of a class by clicking on the cross-hairs button, as shown in Figure 3-127 on page 3-146.

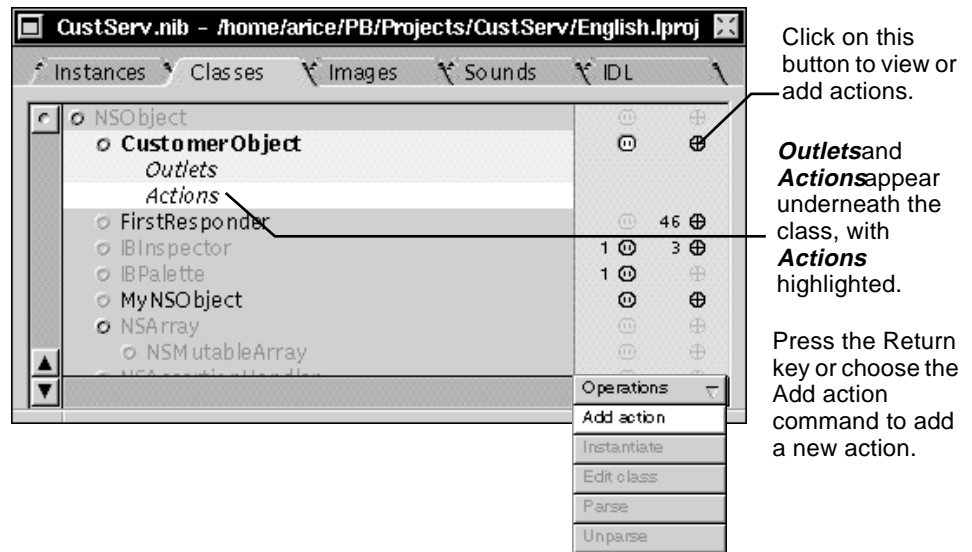


Figure 3-127 Accessing the Actions of a Class

When you press the Return key or choose the Add action command from the Operations menu, a new action is displayed under **Actions**, as shown in Figure 3-128. Type the name of the action in place of the default name and press the Return key.

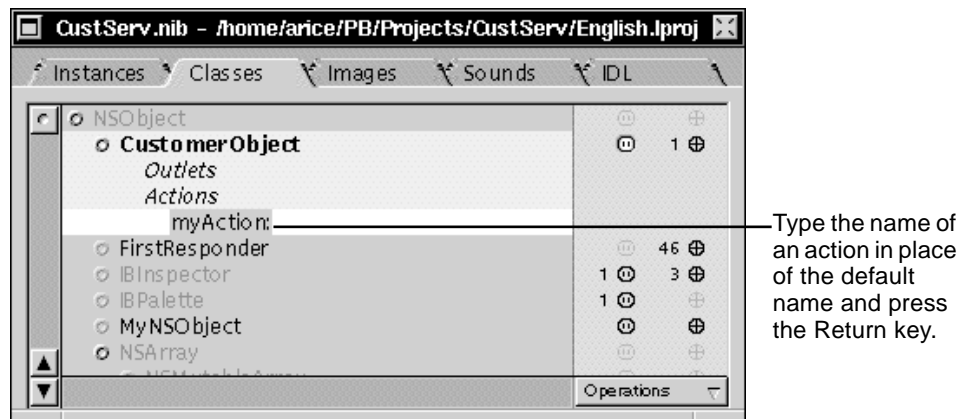


Figure 3-128 Naming a New Action

When you press the Return key, the action is renamed and Interface Builder highlights the new action. If you did not specify a colon (:) after the action name, Interface Builder appends it for you. To add another action, press the Return key again.

When you have finished adding outlets and actions, click on the class name to collapse the list of outlets and actions.

Creating an Instance of Your Class

You cannot connect classes to other classes. Only instances of classes—objects—can really communicate with each other. Interface Builder requires a real instance of your class to enable the connection of your object to other objects.

The procedure for generating instances of non-`NSView` classes in Interface Builder is simple (see Figure 3-129). The following steps apply only to classes that do not inherit from the `NSView` class.

- 1. Select your class in the Classes display.**
- 2. Choose Instantiate from the Operations pull-down menu.**

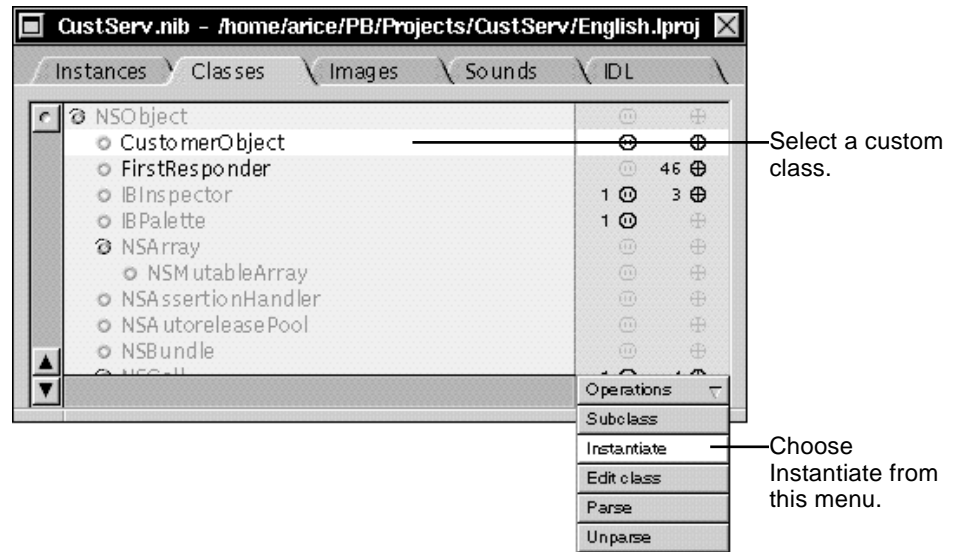


Figure 3-129 Instantiating a Custom Class

When the new instance is displayed in the Instances display (see Figure 3-130 on page 3-149), it takes the same name as the class. Rename it, if you want, to something more indicative of an object. Double-click on the text to select it, then type the new name. For example, `AppController` could become `AppControllerObject`. Be aware, however, that this name is merely a convenient way to identify the object in Interface Builder; it does not create an identifier that you can reference in code.

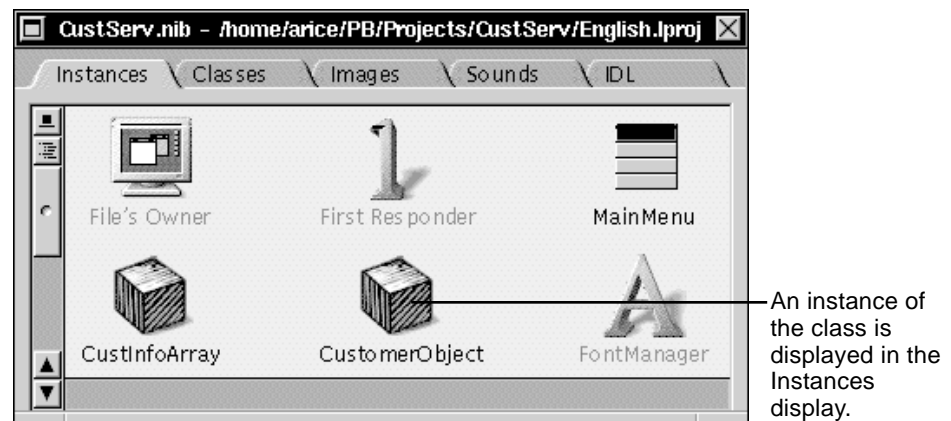


Figure 3-130 The New Instance in the Instances Display

Connecting Your Class's Outlets

An outlet is an instance variable that identifies another object. You initialize an outlet in Interface Builder by making a connection from your instance to another object, as shown in Figure 3-131 on page 3-150. To do so, perform the following steps:

1. **Control-drag a connection line from the instance to another object.**
2. **In the Connections display, select the outlet that identifies the destination object.**
3. **Click on the Connect button.**

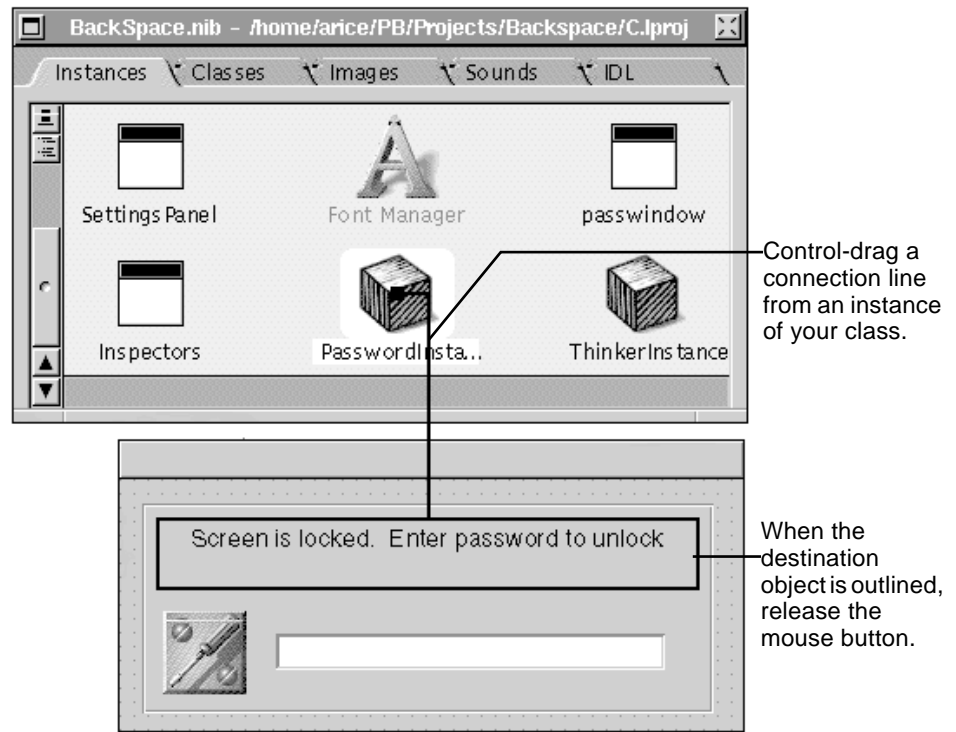


Figure 3-131 Connecting an Outlet

When you establish the line connection, the Inspector panel for the destination object becomes the key window. Specify the outlet identifier for this object as shown in Figure 3-132 on page 3-151.

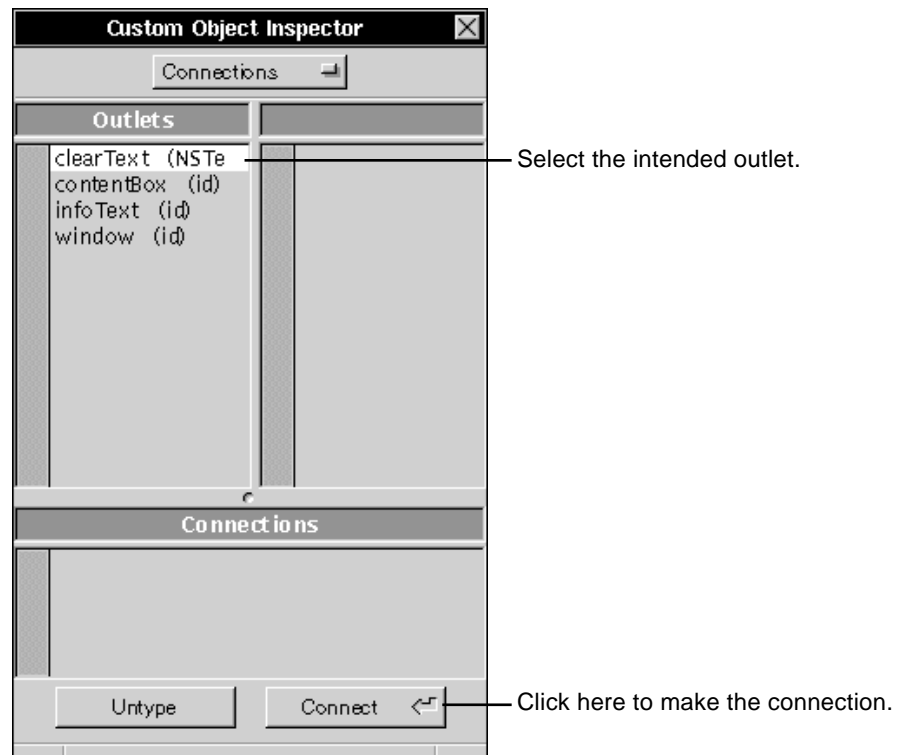


Figure 3-132 Specifying the Outlet Identifier

Note – This task and the next one, “Connecting Your Class’s Actions,” summarize information presented more fully in “Making and Managing Connections” on page 3-109.

Connecting Your Class’s Actions

An action is a method that an `NSControl` object invokes in your instance—the target object—when a user activates the `NSControl` (for example, clicks on a button). you make an action connection in Interface Builder by drawing a connection line form the `NSControl` object to the instance of your class, as shown in Figure 3-133 on page 3-152.

To make an action connection, do the following:

1. **Control-drag a connection line from an NSControl object to your class's instance.**
2. **In the Connections display, select the appropriate action.**
3. **Click on the Connect button.**

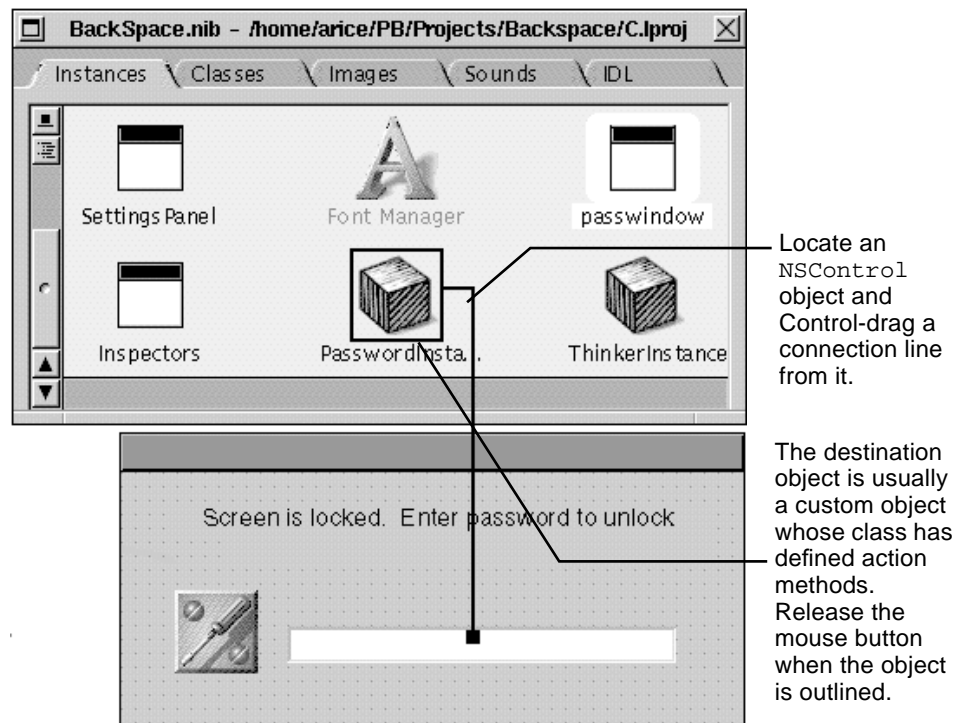


Figure 3-133 Connecting an NSControl Object

When the line is set between objects, the second column of the Connections display shows the action methods that the target object (your instance) has declared. Select the action for this NSControl object, as shown in Figure 3-134 on page 3-153.

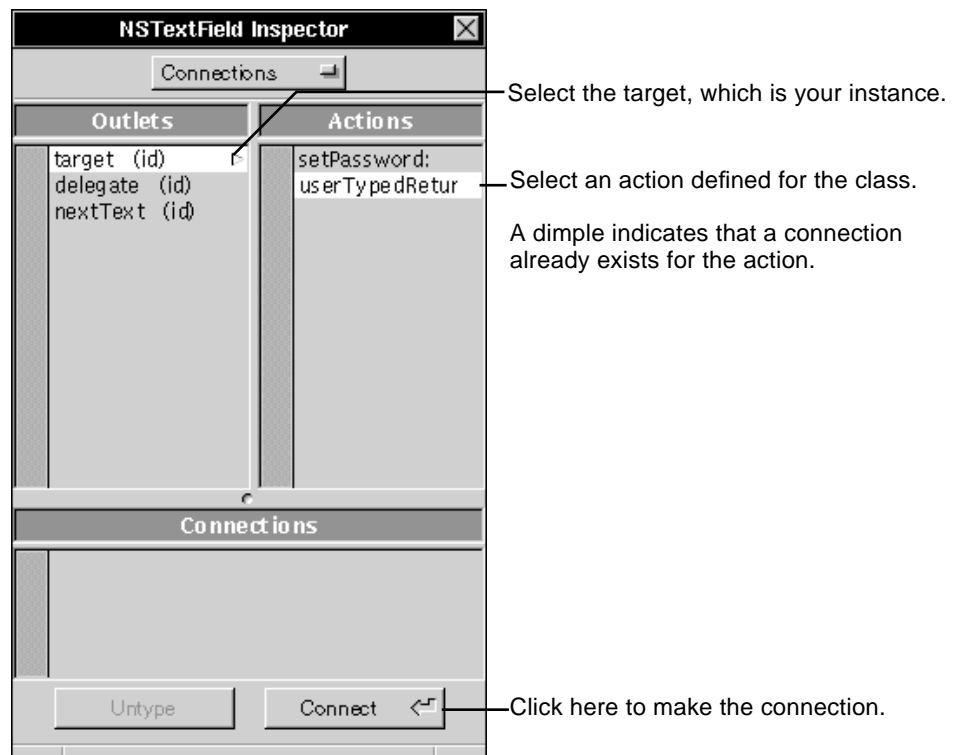


Figure 3-134 Selecting the Action Method

Note – You can make connections between objects entirely within the outline mode of the Instances display. For more information on the outline mode, see “Making and Managing Connections” on page 3-109.

Generating Source Code Files

Before you begin specifying the behavior of your class in code, you typically generate template source code files for your class from the information contained in the nib file. The header file *MyClass.h* created by Interface Builder declares the outlets you specified as instance variables of type `id` and

declares the actions as instance methods of the form *methodName.sender*. The implementation file *MyClass.m* contains empty function blocks for each of these methods.

To generate source code files for your class, do the following:

1. **Select your class in the Classes display.**
2. **Choose Unparse from the Operations pull-down menu.**
3. **Click on Yes in the subsequent attention panels.**

Interface Builder generates template code files by unparsing the nib file (see Figure 3-135 on page 3-154).

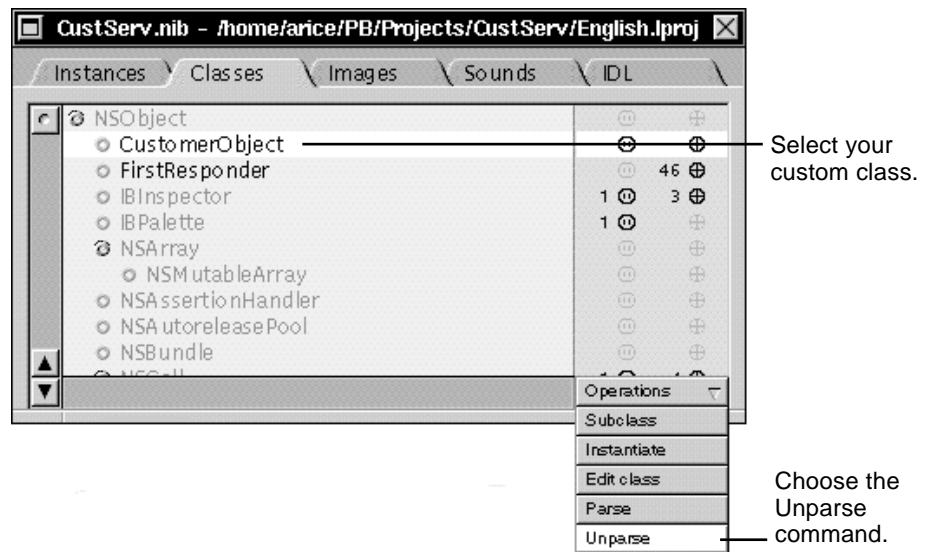


Figure 3-135 Unparsing the Nib File

Interface Builder then displays an attention panel to confirm creation of the files (see Figure 3-136 on page 3-155).

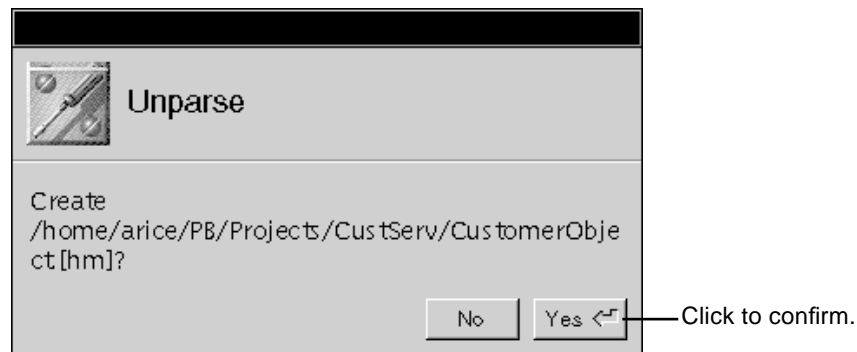


Figure 3-136 Unparse Attention Panel

If you confirm creation and the nib file is associated with a project, another attention panel subsequently asks if you want to add the template code files to the project. Click on Yes to add the files to the project.

Implementing a Subclass of `NSObject`

This task summarizes the steps you must complete—and can optionally complete—to implement a subclass of `NSObject`. With this kind of subclass, the subtleties arising from inherited behavior are simplified. Still, the interaction of your class with the root class is very important, and applies to all subclasses.

In this task you write code, and so there is a temporary departure from interface Builder. The task assumes you have complete the following prerequisites in Interface Builder, presented earlier in this chapter:

- Naming a class and positioning it in the class hierarchy
- Specifying outlets and actions for the class
- Creating an instance of the class
- Connecting the instance to other objects through the outlets and actions
- Generating code files by unparsing the nib file

When you have generated code files in Interface Builder, switch over to Project Builder and open your project. Open your class's header file (`ClassName.h`) and implementation file (`ClassName.m`) in Edit windows.

- 1. Import header files**
- 2. Declare new instance variables.**
- 3. Implement accessor methods.**
- 4. Define target/action behavior.**
- 5. Define initialization and cleanup behavior.**
- 6. Define how objects are copied.**
- 7. Define how objects are compared.**
- 8. Implement archiving and unarchiving.**
- 9. Define special behavior for your class.**

Making Your Class a Delegate

Several OpenStep classes allow you to register their object as a delegate. As certain events occur, the kit objects send messages to their delegates, giving them the opportunity to participate in processing. In Interface Builder, you can easily designate your class's instance as a delegate, as shown in Figure 3-137 on page 3-157. To do so, perform the following steps:

- 1. Connect your instance to an object that has delegates.**
- 2. Select the delegate outlet in the Connections Inspector panel.**
- 3. Click on Connect.**
- 4. Implement the delegate methods.**

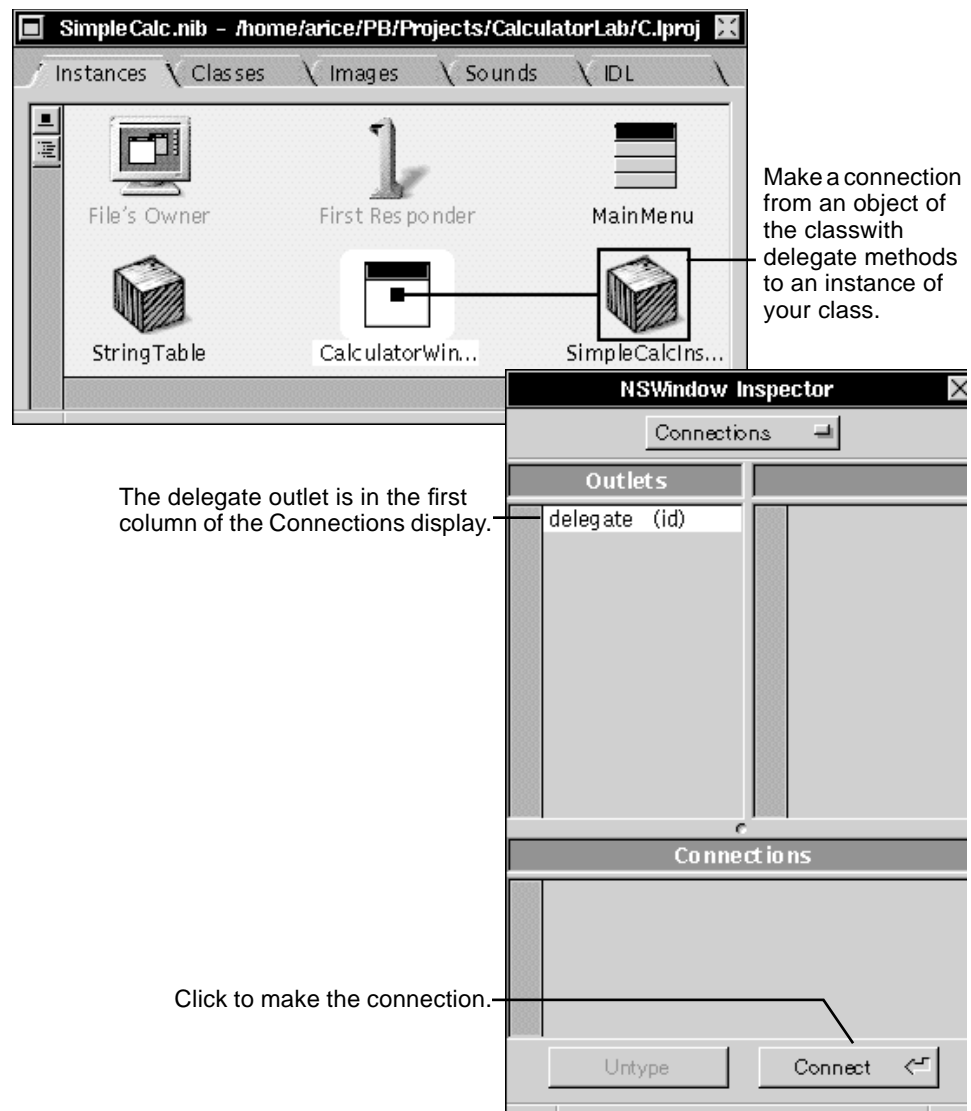


Figure 3-137 Making Your Class a Delegate

Next implement the delegate methods to which you want your object to respond.

Note – Messages to delegates sometimes notify them of impending or just-transpired events, and sometimes request them to complete some work. Major classes with delegate methods are `NSApplication`, `NSWindow`, `NSText`, and `NSBrowser`. See *OpenStep Programming Reference* for details on delegate methods

Implementing an `NSView` Subclass

Making a subclass of the `NSView` class is a procedure that differs from making a subclass of the `NSObject` class. But it starts in the same way. In the Classes display of the nib file window, choose Subclass from the Operations pull-down menu while `NSView` is highlighted in the browser. Then name your class and add outlets and actions.

To implement an `NSView` subclass, do the following:

- 1. Identify the class and its outlets and actions.**
- 2. Place and resize and `NSCustomView` object on a window or panel.**
- 3. Assign your class as the class of the `NSCustomView`.**
- 4. Connect the instance to other objects in the interface.**
- 5. Generate code files.**
- 6. Complete programming tasks necessary for any object.**

7. Complete programming tasks specific to `NSView` objects.

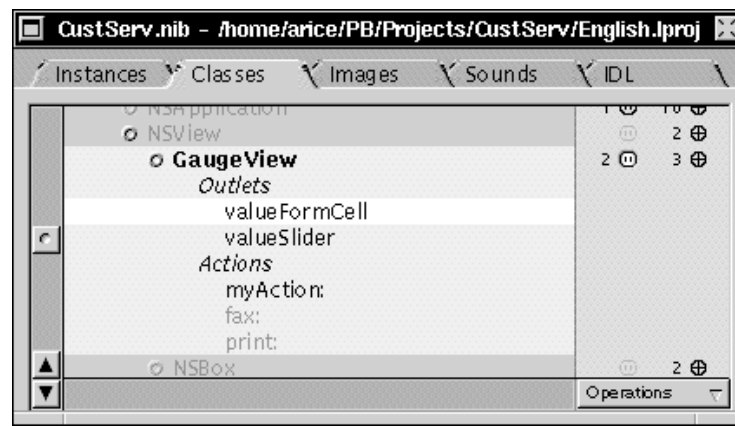


Figure 3-138 An `NSView` Custom Class

Note – The steps in this task, insofar as they apply to `NSView`, also apply to creating classes that inherit from subclasses of `NSView`.

Place a proxy instance of your class in your interface, as shown in Figure 3-139 on page 3-160. Interface Builder provides a `CustomView` object to represent instances of `NSView` subclasses.

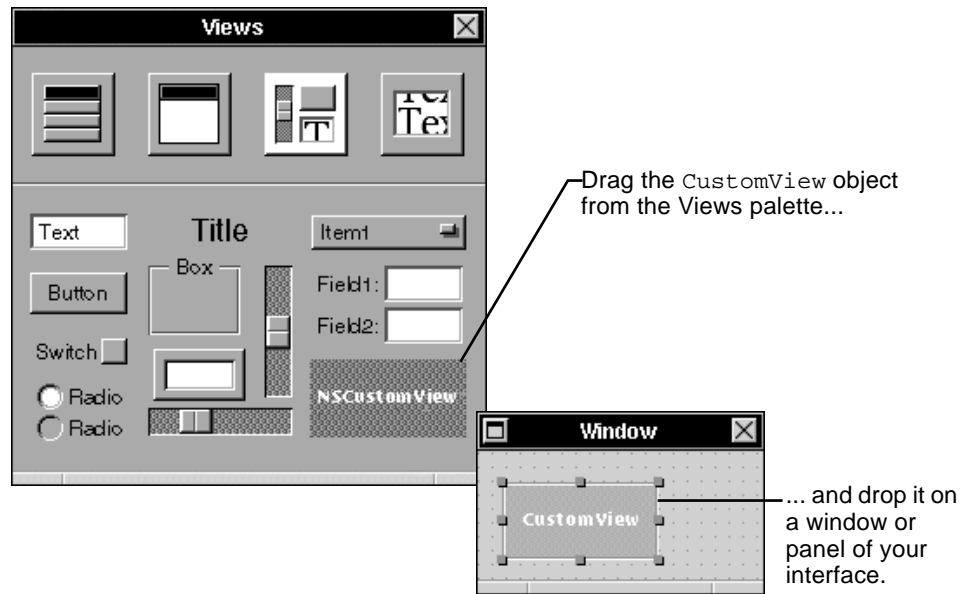


Figure 3-139 Making an Instance of an NSView Subclass

Position and resize the `CustomView` object, and while it is still selected, bring up the Attributes display of the Inspector panel (see Figure 3-140 on page 3-161). Assign a class name to the object; this creates an instance of your `NSView` subclass.

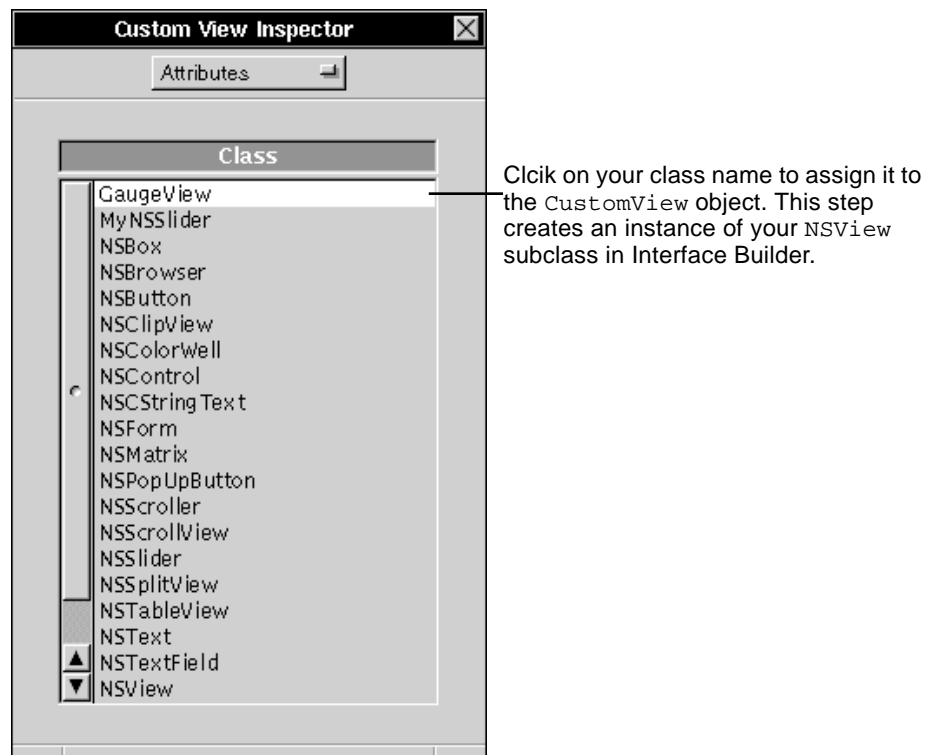


Figure 3-140 Assigning a Class Name to your NSView Object

Now complete the following three tasks, which are the same tasks that follow the instantiation of an NSObject subclass:

- Connect the instance to other objects in the interface (see “Connecting Your Class’s Outlets” on page 3-149 and “Connecting Your Class’s Actions” on page 3-151). But now the instance is displayed as part of the interface, and not as an icon in the Instances display of the nib file window.
- Generate code files and have them inserted in your project (see “Generating Source Code Files” on page 3-153).
- Switch over to the project in Project Builder that contains the nib file. Open your class’s code files in Edit and complete the programming tasks for your subclass.

Adding Existing Classes to Your Nib File

- Drag the header file from the File Viewer or Project Builder into the nib file window (see Figure 3-141).

or

- Copy a class in one nib file and paste it into another.

The easiest way to add a class to your nib file is to drag the header file for an existing custom class from the Workspace Manager's File Viewer into Interface Builder.

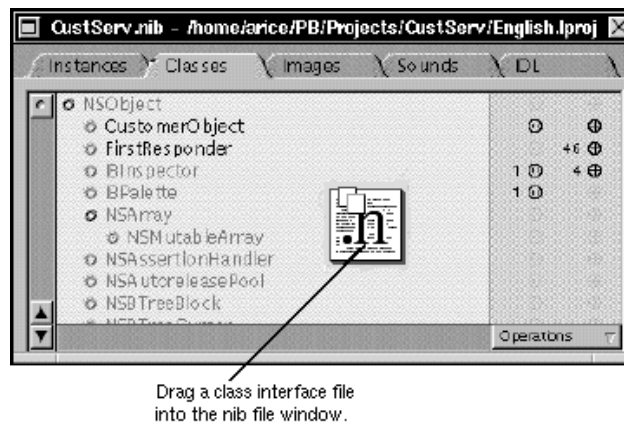


Figure 3-141 Dragging a Header File into Your Nib File

You can also add a class definition to a nib file by dragging a header or implementation file from Project Builder into a nib file window.

The new class is displayed in the Classes display under its subclass and with its outlets and actions defined. After adding the class, you must still connect it to other objects through its outlets and actions. To do so, complete the following steps:

1. **Make an instance of the class (for NSView subclasses, that means assigning your class name to the NSCustomView object).**
2. **Connect the instance's outlets and actions to other objects in the nib file.**

Note – Instead of defining a class in Interface Builder, you can write a header file and drag it into a nib file window as described above. When writing your header file, be sure to declare outlets as instance variables of type `id`. Declare actions as methods with a single argument: `sender`.

Updating a Class Definition

If you later add outlets and actions to the header file, or delete them from it, Interface Builder allows you to update the nib file with this new information (see Figure 3-142).

- Choose the Parse command and select a header file in the Open panel.

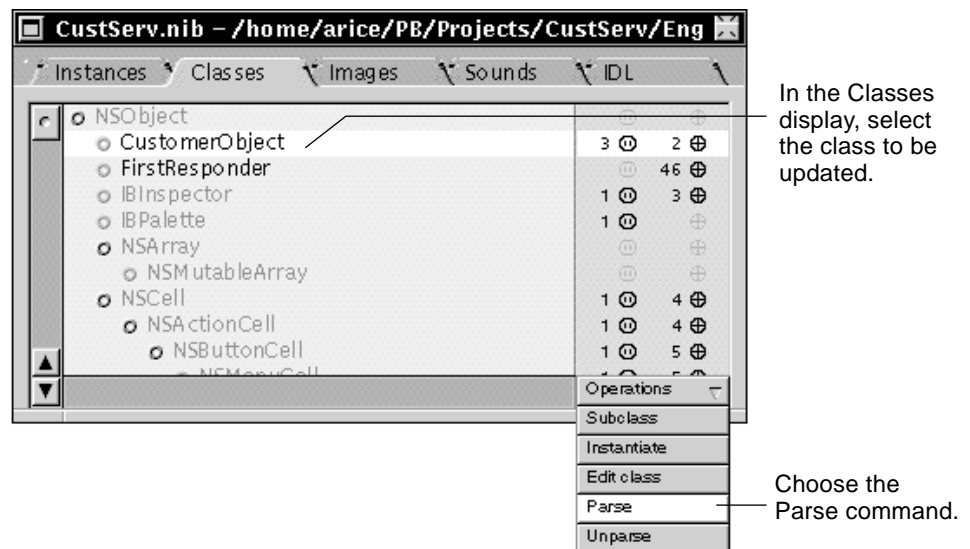
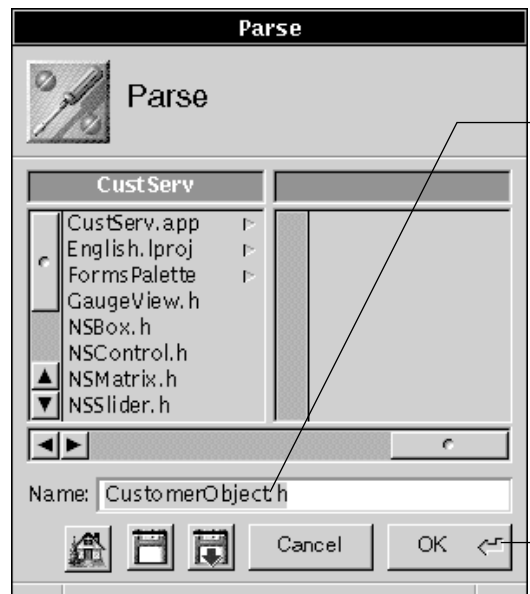


Figure 3-142 Updating the Nib File

Interface Builder brings up an Open panel for you to confirm (or select) the class definition to update (see Figure 3-143 on page 3-164).



The header file of the class selected in the Classes display is highlighted. If you did not select a class in the Classes display, select one now or type its name.

Click to have the new information parsed into the nib file.

Figure 3-143 Selecting the Class Definition to Update

If there are any new outlets and actions, remember to connect these outlets and actions to other objects in the nib file.

Note – You can also use the Parse command to add an existing class to a nib file (see “Adding Existing Classes to Your Nib File” on page 3-162), or you can create a header file and read it into a nib file through the Parse command.

Adding IDL Template Objects to Your Interface

The IDL display of the File window shows the IDL types that are available to your application. You can parse IDL files into your application and instantiate IDL template objects using the Actions pull-down list.

The Parse IDL button (see Figure 3-144) displays an Open panel that lets you specify the IDL type file you want Interface Builder to parse. Interface Builder parses the type from the file and then displays the names of the IDL types in the IDL display.



Figure 3-144 Parse IDL Button (IDL Display)

You can instantiate an IDL template object from any of the IDL interfaces shown in the IDL display.

The Make Template Object button (see Figure 3-145) creates an IDL template object and places an icon representing that object in the Instances display of the File window. The template object does not make a connection with its NEO system until the nib file is loaded into your application at run time. Therefore, the NEO system does not have to be running when you create interfaces with IDL template objects.

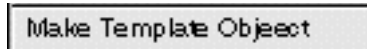


Figure 3-145 Make Template Object Button (IDL Display)

The name of the object is derived from the name of the IDL type from which it is instantiated. You must change its name in your application to the name by which it is known to the NEO Naming Service. If you have access to the NamingContext (that is, the server that implements the object is running and is part of your NEO workgroup), you can determine this name by using `neoadmin`. (For an overview of the NEO Naming Service and information on using `neoadmin`, see *NEO System Management*.) If you do not have access to the NamingContext, you may need to get the correct name for the object from the developer who developed its server. You change the name by bringing up an Attributes inspector for the IDL template object and typing in the new name.

Connecting IDL Template Objects

Connections cannot be dragged from IDL template objects because IDL template objects have no outlets or actions to which connections can be made. Connections can be dragged from `NSCustomObjects` or `NSCustomViews` to IDL objects because the outlets in `NSCustomObjects` and `NSCustomViews` that you add can have their types changed (see "Outlet Autotyping" below).

After you click on the Connect button, the button's title changes to Disconnect, allowing you to remove the connection. If you cut or copy a connected object and then paste it, its connections are severed.

Note – You can connect IDL template objects only to `NSCustomObjects` or `NSCustomViews` subclassed from Application Kit objects. You must use outlets that you have added to the custom objects. See “Outlet Autotyping” below for information on how outlets are autotyped when connected to IDL template objects.

Outlet Autotyping

The OpenStep programmatic interface encourages developers to add types to variables wherever possible, but does not require types. Interface Builder supports types by autotyping outlets in `NSCustomObjects` and `NSCustomViews`.

When you connect an object to an outlet you have added to an `NSCustomObject` or `NSCustomView`, the outlet's type is automatically changed from `id` to the class of the object that is the source of the connection. If the source object is an IDL template object, the IDL type is used instead.

For Objective C objects, autotyping reduces the number of warning messages emitted by the compiler when compiling code generated by Interface Builder, and reduces the possibility of a run time type error (that is, an exception due to the object not supporting a method selector). For IDL template objects, autotyping removes the possibility of a compiler error, since IDL types are statically checked and the compiler does not compile a file unless the IDL types of objects and variables match.

If you later try to connect the same outlet to an object of a different class, the outlet type may be altered:

- If the source object's class is a subclass of the outlet type, then the connection succeeds and the outlet type is not changed for either IDL or Objective C.
- If the source object's class is a superclass of the outlet type, then the outlet's type is changed to the source object's class for both IDL and Objective C.

- If either the source object or outlet type is an Objective C class, and the other is an IDL type, you are prompted about whether you want to change the outlet type, removing all existing connections to other objects. If you answer YES, the outlet type is changed to the source object's type, and all connections to other objects having that outlet are removed.
- If no subclass/superclass relationship exists, then an outlet with an Objective C type is changed to `id`. You are prompted about whether you want to change the outlet type for an outlet with an IDL type, removing all existing connections to other objects. If you answer YES, the outlet type is changed to the source object's type, and all connections to other objects having that outlet are removed.

In addition, the outlet browser in the Connections display of the Inspector panel deactivates outlets and the Connect button if a connection is dragged to an object that cannot be connected to the outlet. For example, an IDL object cannot be connected to the `target` outlet in an `NSButton` object, because an `NSButton` object is created by the Application Kit and the types of Application Kit outlets cannot be changed (they are always `id`).

Setting Preferences

You open the Preferences panel by choosing the Preferences command in the Info menu. This panel has two displays; you use the pop-up list at the top of the panel to access these displays.

General Preferences

These preferences control which panels appear when Interface Builder is launched and also whether a backup file is created when the nib file is saved.

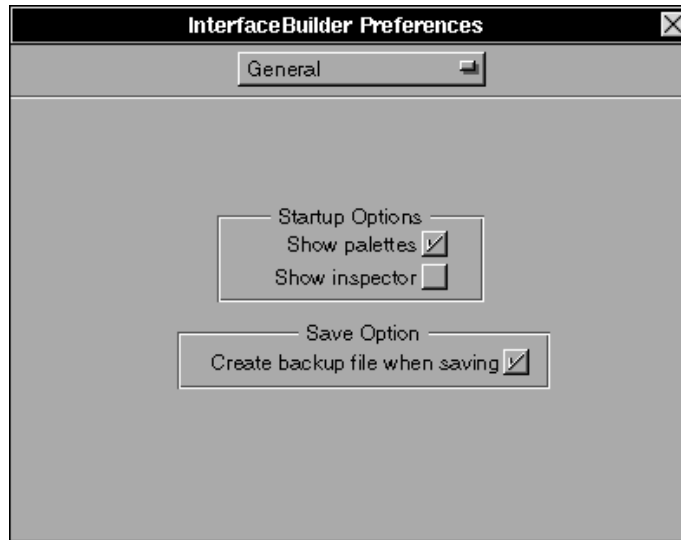


Figure 3-146 Interface Builder’s General Preferences Panel

If the `Save Option` box is checked, Interface Builder creates a backup file whenever you save a nib file that has been modified. Assuming the box is checked, if you open a nib file named `FindPanel.nib`, make changes, and then save the modified file, Interface Builder renames the original file `FindPanel.nib~` before saving the modified file as `FindPanel.nib`. Because of the safety of having a backup file, it is generally better to leave this box checked.

Palettes Preferences

This display of the Preferences panel shows you which palettes are available to Interface Builder and lets you control which palettes are installed in the Palettes window.

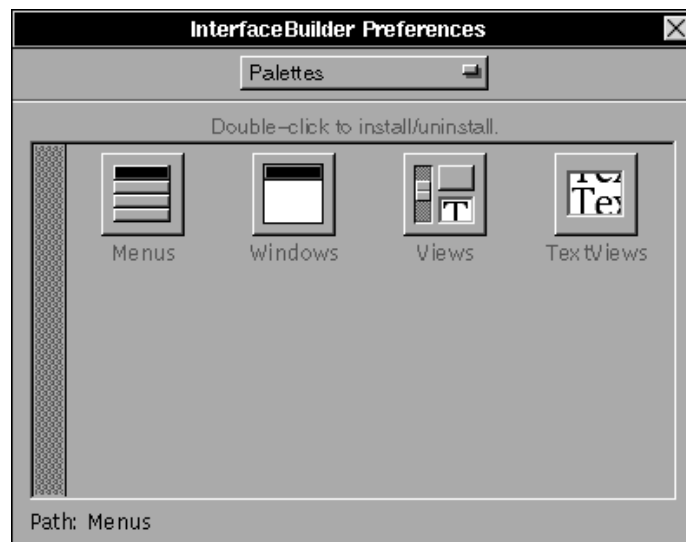


Figure 3-147 Interface Builder's Palettes Preferences Panel

Each palette is represented by an icon. The palettes that are already installed in the Palettes window display their titles in gray; those that have not been installed display their titles in black. Double-clicking on the icon toggles the state of the palette: If it was installed, it is removed from the panel; if it was uninstalled, it is installed in the panel.

When Interface Builder begins running, it loads the standard palettes (those displayed in the top row of the illustration above). It also adds to the Preferences display any palettes the user has previously loaded using the `Open . . .` command on the Palettes menu.

Adding Custom Palettes, Inspectors, and Editors

Interface Builder's primary value as a development tool is that it lets you interact directly with the objects that will make up your application. In general, these objects are defined by the OpenStep system software. However, it is possible to extend Interface Builder's library of objects by creating *custom palettes*, thus letting you interact directly with objects that you or other developers have created.

A custom palette can contain objects of various sorts. Most commonly, a custom palette contains `NSView` objects, objects that the user instantiates by dragging into a standard window. It is also possible to create custom palettes that contain `NSMenuCells` (which are instantiated by being dragged into a menu), `NSWindows` (which are instantiated by being dragged into the workspace), and other non-`NSView` objects (which are instantiated by being dragged into the File window).

For any custom palette object, you can provide one or more inspectors. A custom object's inspector is displayed in the Inspector panel when the user selects the object. Most custom objects require an Attributes inspector. For example, the fictitious `RepeatButton` class mentioned earlier would probably require an Attributes inspector to let the user set the repeat rate for a given button. It could also supply its own Connections, Size, and Help inspectors, although the standard versions of these inspectors are generally adequate for most uses.

Finally, a more complex custom object may require its own *editor*. An editor controls how a user can interact with a selected object. Interface Builder itself supplies editors for the objects it knows about. For example, when you double-click on a window icon in the File window, Interface Builder's window editor is invoked and brings the actual window to the front. Or, when you double-click on an `NSForm` object in an application window, Interface Builder's matrix editor is invoked, letting you drag cells to new positions.

An editor that you provide must open its own window when the user double-clicks on the custom object. Since each custom object can have its own editor window, editors make copy and paste or drag and drop operations between editor windows possible.

Creating custom palettes, inspectors, and editors involves working with Interface Builder's application programming interface (API). This API is described in detail in Appendix B, "Interface Builder Application Programming Interface," Appendix C, "Interface Builder API Classes," Appendix D, "Interface Builder API Protocols," and Appendix E, "Interface Builder API Types and Constants."

Interface Builder Command Reference

The remainder of this chapter gives short descriptions of Interface Builder's commands. Only those commands that are unique to Interface Builder are listed; for information on commands that are common to all OpenStep applications see *Using the OpenStep Desktop*.

Commands in the Document Menu

These commands act to open, create, save, or test an Interface Builder document. (Interface Builder documents are generally referred to as “nib files,” since that is how they are stored on disk. However, until a document is saved, no file exists, so referring to the document as a “nib file” is not strictly correct. Even so, for simplicity, Interface Builder documents are referred to as nib files throughout, unless to do so would cause confusion.)

Table 3-9 Document Menu Commands

Command	Description
Open	Opens an existing nib file.
New Application	Creates a new Interface Builder nib file containing the basic components of an application: a main menu, a standard window, and other resources. You rarely use this command since it is generally more convenient to have Project Builder create the nib file for a new application. See the description of Project Builder's New command in “Commands in the Project Menu” on page 2-33 for more information.
New Module	Opens the New Module submenu, which offers commands for creating various sorts of Interface Builder nib files other than the type used for an application's main nib file. See “Commands in the New Module Submenu” on page 3-172 for more information.
Save	Saves the current nib file. You can edit more than one Interface Builder nib file at a time. Each open nib file is represented by a File window. The File window that has main or key window status identifies the current nib file.
Save As	Saves the current nib file under a different file name.

Table 3-9 Document Menu Commands (Continued)

Command	Description
Save All	Saves all open nib files.
Revert to Saved	Restores the current nib document to the state represented in the nib file. All changes made since the file was last saved are lost.
Test Interface	Puts Interface Builder in test mode. When you choose this command, Interface Builder’s supporting windows disappear, leaving only those windows that belong to your application. You can then test the operation of the objects in your application. See “Testing the Interface” on page 3-136 for more information.

Commands in the New Module Submenu

These commands let you create auxiliary nib files of various sorts. (The main nib file is generally created by Project Builder.)

Table 3-10 New-Module Menu Commands

Command	Description
New Empty	Creates the simplest sort of nib file, one that includes references only to a File’s Owner object and a First Responder object.
New Info Panel	Creates an auxiliary nib file containing a panel that is preconfigured as a standard Info panel.
New Attention Panel	Creates an auxiliary nib file containing a panel that is preconfigured as a standard Attention panel.
New Inspector	Creates an auxiliary nib file containing the components you need when creating an inspector for a custom palette project.
New Palette	Creates an auxiliary nib file containing the components you need for a custom palette project. You rarely issue this command directly, since Project Builder provides this nib for you when you create a new palette project.

Commands in the Edit Menu

Except for the `Set Name` command, this menu contains the standard editing commands: `Cut`, `Copy`, `Paste`, `Delete`, and `Select All`. These commands work in the expected ways.

Table 3-11 Edit Menu Commands

Command	Description
Enter selection	Highlights the name of the selected object in the outline mode of the nib file window's Instances display. To use this command, select an object in the interface, then choose Enter Selection. The appropriate object line in the outline mode is highlighted. This command is useful when you are using outline mode and you want to know which line in the outline mode represents a particular object in the interface.
Set Name	Displays a panel that lets you set the name of the selected object. With this name, and the <code>NSGetNamedObject()</code> function, you can access objects by name within your application. However, it is generally a better idea to access objects through the use of outlets, since outlets can be connected and disconnected in Interface Builder, eliminating the need to alter your application's code.

Commands in the Format Menu

This menu lets you set the font and formatting attributes of the selected object. It also gives you access to the `Group` submenu, the `Align` submenu, the `Size` submenu, and the `Page Layout` panel.

Table 3-12 Format Menu Commands

Command	Description
Font	Opens the Font submenu. Interface Builder's use of the Font submenu is entirely standard. By setting the font of an NSTextField or the NSText object within the NSScrollView (for example), you are determining which font the user will use in those objects when the application runs.
Text	Opens the Text submenu. Interface Builder's use of the Text submenu is entirely standard. By setting the text alignment or tab settings of an object in Interface Builder, you are determining the alignment and tab settings for those objects at run time. Note that the ruler commands work only with a Text object that is the document view of an NSScrollView.
Bring To Front	Establishes the selected object as the frontmost object in the window. If the selected object intersects other objects, the selected one is drawn over the others. If more than one object is selected when you choose this command, the entire group of objects is brought in front of all other objects in the window.
Send To Back	Puts the selected object or objects behind all other objects in the window.
Group	Opens the Group submenu, which is described in "Commands in the Group Submenu" on page 3-175.
Align	Opens the Align submenu, which is described in "Commands in the Align Submenu" on page 3-175
Size	Opens the Size submenu, which is described in "Commands in the Size Submenu" on page 3-177
Page Layout	Opens the standard Page Layout panel. This panel lets you specify how the window you print using Interface Builder's Print command will appear on paper. Since a screen pixel is approximately 75 percent of the size of a printer pixel, the image of a window looks larger on paper than it does on the screen. To compensate, set the scaling factor to 75 percent in the Page Layout panel's Scale field.

Commands in the Group Submenu

This menu offers commands that help you group the selected object or objects.

Table 3-13 Group Submenu Commands

Command	Description
Group	Puts the selected object or objects behind all other objects in the window.
Group in ScrollView	Groups the selected object or objects in an NSScrollView. The NSScrollView is sized so that it just accommodates the objects in the group. The grouped objects are made subviews of the NSScrollView.
Group in SplitView	Groups the selected objects within an NSSplitView object. The objects become subviews of
Ungroup	Removes the grouping established by the Group or Group in NSScrollView commands.

Commands in the Align Submenu

This menu provides commands that let you position objects accurately within a window.

Table 3-14 Align Submenu Commands

Command	Description
Set Grid On / Off	<p>Enables and disables the alignment grid in all windows of all open nib files. When the grid is enabled, View objects dragged into a window are constrained in their location and dimensions to the units defined by the grid.</p> <p>By default, the intersections of the grid are aligned, both vertically and horizontally, on every eighth pixel in a window. Also by default, an object's lower left corner is the reference point for alignment with the grid. (The grid spacing and the object's reference point can be changed using the Alignment panel.)</p> <p>By default, the intersections of the grid are aligned, both vertically and horizontally, on every eighth pixel in a window. Also by default, an object's lower left corner is the reference point for alignment with the grid. (The grid spacing and the object's reference point can be changed using the Alignment panel.)</p>
Show Grid / Hide Grid	<p>Displays and hides the alignment grid in all windows of all open nib files. The grid is displayed as a rectangular array of dark gray dots.</p>
Align To Grid	<p>Aligns the selected objects to the nearest grid mark. To use this command, first select one or more objects by dragging around them or by Shift-clicking on them..</p>

Table 3-14 Align Submenu Commands

Command	Description
Make Row	Aligns the selected objects horizontally. The row extends to the right of the selected object that is nearest the top left corner of the window. The spacing between objects is determined by the original spacing between the two objects nearest the window's top left corner. If these objects originally overlapped, the objects in the resulting row abut each other.
Make Column	Aligns the selected objects vertically. The column forms below the object that is nearest the top left corner of the window. The spacing between objects is determined by the original vertical spacing between the two objects nearest the window's top left corner. If these objects originally overlapped, the objects in the resulting column abut each other.
Alignment	Opens the Alignment panel, which is described in the "Using the Alignment Panel" on page 3-45.

Commands in the Size Submenu

This menu's commands allow you to resize an object.

Table 3-15 Size Submenu Commands

Command	Description
Size to Fit	Resizes the selected object to the minimum size required to display its contents. If more than one object is selected, each is resized to its own minimum size. For an object of a given class, minimum size may depend on the font used to display the title, the alignment and location of the title, and the distance the content area is offset from other areas of the object.
Same Size	Forces one or more selected objects to assume the dimensions of another selected object. The first object you select establishes the dimensions that the other selected objects will assume.

Commands in the Tools Menu

This menu's commands open or bring to the front the named panel

Table 3-16 Tools Menu Commands

Command	Description
Colors	Displays the Colors panel.
Inspector	Displays the Inspector panel.
Palettes	Opens the Palettes submenu, which is described in “Commands on the Palettes Submenu” below.
Help Builder	Displays the Help Builder panel. This panel will be empty unless your application is part of a project containing a <code>Help</code> directory. See “Attaching Help to Objects” on page 3-132 for information on using the Help Builder panel.

Commands on the Palettes Submenu

This menu provides commands that let you open, create, save, and close palettes

Table 3-17 Palettes Menu Commands

Command	Description
Open...	Presents an Open panel, enabling you to load additional palettes into Interface Builder's Palette window. See "Adding Custom Palettes, Inspectors, and Editors" on page 3-71 for more information.
New	Creates a dynamic palette. The Plette window will display a blank palette, to which you can add objects from your interface. Dynamic palettes do not have a palette project associated with them, and they do not have to be compiled.
Palettes...	Displays the Palette window. You can always access the four standard palettes using the palette window. You can also load nonstandard palettes into the window using the Open command on the Palettes menu.

Using Edit in Developer Mode



In addition to the standard UNIX editing tools (vi and Emacs), the OpenStep development environment provides a mouse-based text editor named Edit for creating and editing ASCII or RTF (Rich Text Format[®]) text files.

Edit has all the standard features of a text editor: You can type paragraphs of text without pressing the Return key (the text wraps automatically at the end of each line, and if you change fonts or resize the window, the text rewraps accordingly). You can use the mouse to select where text will be entered and to select text you want to edit. And you can find and replace text, move and copy it, and so on.

While Edit has the functionality of a good text editor, it is particularly suited for writing programming code and performing other application-development tasks. It lacks many of the capabilities found in similar applications, but in Developer Mode it has many features specifically designed for programmers. For example, Edit supports name expansion, folder browsing, block nesting in program listings, and a structured editing facility. It also provides interapplication functionality with Project Builder and Terminal.

Starting Edit

You can start Edit from the workspace as you would start any other application. Alternatively, you can start Edit from a shell window by typing the following command at the UNIX prompt:

```
/usr/openstep/Apps/Edit.app/Edit [file name ...] &
```

Several command-line options allow you to override various default characteristics of Edit for the work session you are about to start—characteristics such as the number of lines and columns in new windows, the font family used, and the font size. For example:

```
/usr/openstep/Apps/Edit.app/Edit -NSFont Times-Roman Fruit.m &
```

These command-line options can be specified in any order, as long as they precede any file names. Several options are listed in Table 4-1.

Table 4-1 Edit Command-line Options

Option	Effect
IndentWidth	Specifies the width of indentation for block nesting. The default value is 4.
NSFont	Specifies the font family. The default font is Helvetica.
NSFontSize	Specifies the font size, in points. The default value is 12.
Tags	Specifies one or more path names to <code>tags</code> files that will be searched by the Source command. The path names should be separated by a colon, as in a standard UNIX path list. The default is “tags,” which indicates that the <code>tags</code> file in the current folder will be searched. See the description of using <code>tags</code> files under “Interacting with UNIX” on page 4-21 for more information about using <code>tags</code> files in Edit.
DeleteBackup	Specifies whether the previous version of a file is deleted or retained as a backup when you save changes to the file. The default value is YES, which means that the previous version is deleted. If the previous version is saved as a backup, its name is the same as the original file name, but with a tilde (~) appended to the name.
NSMenuX	Specifies the (positive) distance in pixels from the left edge of the screen to the left edge of the main menu.
NSMenuY	Specifies the (positive) distance in pixels from the bottom of the screen to the top of the main menu.

Edit will use the default value for each option unless you override it with a command-line option. The value specified in the command line will remain in effect only for the work session you are about to start. The next time you use Edit, the defaults go back into effect.

You can set new default values for each of the above characteristics (except for screen coordinates) using the Preferences panel, which is described in the “Setting Preferences.” Most defaults set with the Preferences panel remain in effect until you change them.

Setting Preferences

The Preferences command in the Info menu displays the Preferences panel, shown in Figure 4-1. The Preferences panel lets you set default values for various Edit options. For example, you can set default font properties or specify the size of new windows. To use the features of Edit’s Developer Mode, click on the Developer Mode radio button as shown in Figure 4-1.

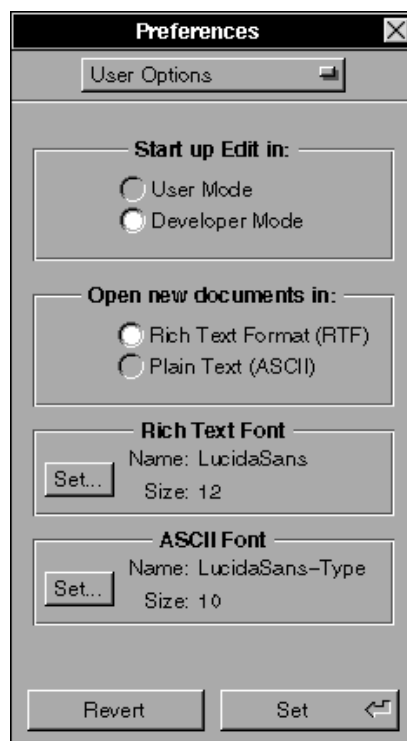


Figure 4-1 Edit Preferences Panel

Enter values and click on buttons to specify new preferences, as described below. Then click on Set to set the new preferences (or click on Revert to restore the previous settings). In general, the new settings remain in effect until you change them. However, you can temporarily override some of the defaults by starting up Edit from a shell window and specifying one or more command-line options (as described in “Starting Edit” on page 4-1).

You can press the button labeled User Options and, in the pop-up list that appears (see Figure 4-2), choose from several other sets of options that are available (see “Global Options” on page 4-6, “Temporary Settings” on page 4-7, “Text Options” on page 4-8, and “C Options” on page 4-11).



Figure 4-2 Options Pop-up List

User Options

Choose User Options in the Preferences panel’s pop-up list to see the user options that you can specify. User options are saved in your defaults database and continue to be used until you specify different values for them.

Start-up Options

Figure 4-3 shows Edit’s start-up options. By default Edit starts in User mode, which presents just a subset of the commands available in Developer mode. If you are using Edit for application development be sure to click on the Developer Mode button.

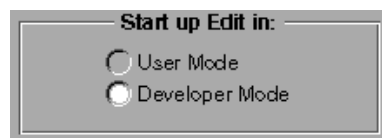


Figure 4-3 Edit Start-up Options

New Document Format Options

Figure 4-4 shows the new document format options. Click on one of the radio buttons to specify whether new documents are created in RTF (Rich Text Format) or ASCII format.

After you have created or opened a document, you can change its format by choosing the Make Rich Text command or the Make ASCII command in the Text menu.

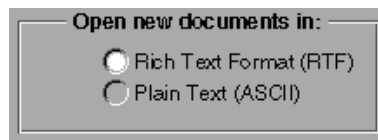


Figure 4-4 New Document Format Options

Default Font for RTF Files

Figure 4-5 shows the field that allows you to set a default font for Edit windows that are in RTF format. Click on the Set button to bring up the Font panel. Specify the font family, typeface, and size, and click on the Set button in the Font panel when you are finished. After you save these settings, all subsequently created RTF documents display text in the specified font by default.



Figure 4-5 RTF Default Font

Default Font for ASCII Files

Figure 4-6 on page 4-6 shows the field that allows you to set a default font for Edit windows that are in ASCII format. Click on the Set button to bring up the Font panel. Specify the font family, typeface, and size, and click on the Set

button in the Font panel when you are finished. After you save these settings, all subsequently opened Edit windows that contain ASCII files display text in the specified font.

When working with code or UNIX command output, it is best to use a fixed-width font family, such as Courier.

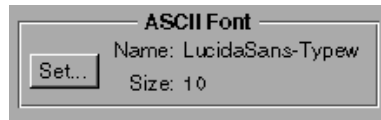


Figure 4-6 ASCII Default Font

Global Options

Choose Global Options in the Preferences panel's pop-up list to see the global options that can be specified. Global options are saved in your defaults database and continue to be used until you specify different values for them.

Save Options

Figure 4-7 shows the save options that determine what happens when you save a file. When you select the Delete backup file option, Edit automatically deletes the previous version of a file when the current version is saved. Click on Don't delete backup file to retain the previous version of a file when you save the current version (if the previous version of a file is saved). This backup file is saved under the original file name, but with a tilde (~) appended to the name.

If you try to save a file that is write-protected, you can do so by responding affirmatively to the confirmation panel that appears as long as you own the file. Check the Save Files Writeable button if you want such write-protected files to lose their write-protected status when they are saved.

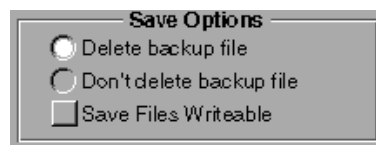


Figure 4-7 Save Options

Default Window Size Options

Figure 4-8 shows the options that let you set a default size for Edit file windows. Enter a width in number of characters in the Width field and a height in number of lines in the Height field. Edit files that you open after saving these settings are displayed in windows with the dimensions you specify. Since these dimensions are specified in characters and lines, the default window sizes are affected by the default font.

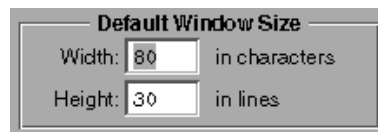


Figure 4-8 Default Window Size Options

Emacs Key Bindings

Figure 4-9 shows the options for Emacs key bindings. Click on one of the radio buttons to specify whether or not Emacs key bindings are enabled.

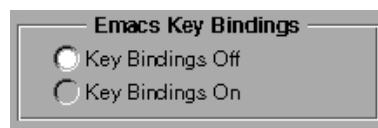


Figure 4-9 Emacs Key Bindings Options

Temporary Settings

Choose `Temporary Settings` in the Preferences panel's pop-up list to see the temporary settings that can be specified. These are called temporary settings because they are not saved in your defaults database.

Line Wrap Options

Figure 4-10 shows the options that determine how text wraps. When you select the Word boundaries option, text wraps onto the following line at the end of each full line, but no words are split across lines. Clicking on Character boundaries also causes text to be wrapped at the end of each line, but words can be split across lines. Clicking on Don't wrap causes text to not wrap at all.

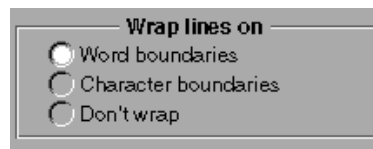


Figure 4-10 Line Wrap Options

Rich Text Display Options

Figure 4-11 shows the options for how RTF files are displayed. When you select the Edit Rich Text Format option, RTF files that you open are displayed as formatted text. Click on Ignore Rich Text Format to view RTF files as unformatted text with the format commands visible. Because other applications use Edit to view formatted text, you should normally leave the Edit Rich Text Format option selected.

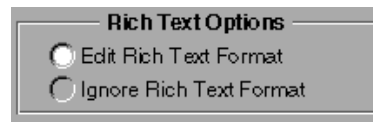


Figure 4-11 Rich Text Display Options

Text Options

Choose Text Options in the Preferences panel's pop-up list to see the text options that you can specify. Text options are saved in your defaults database and continue to be used until you specify different values for them.

Automatic Indenting Options

Figure 4-12 shows the options that determine whether or not lines are automatically indented. When you select the Automatically indent lines option, Edit indents each new line the same as the line above it (automatic indentation is useful for typing indented lines of code). Click on Don't auto-indent lines if you want each new line to start at the left margin.

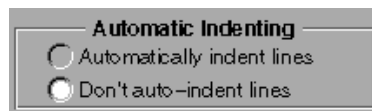


Figure 4-12 Automatic Indenting Options

Structure Level of Blank Lines

Figure 4-13 shows the options that determine how blank lines are treated in your text structure. When you select the Same as previous line option, Edit assigns each “blank” line (that is, each line that contains no visible text) the same structure level as the previous line. Click on Determined by indentation if you want the structure level of blank lines to be determined by the amount of indentation (that is, tabs and spaces) on that line, rather than by the indentation of the previous line.

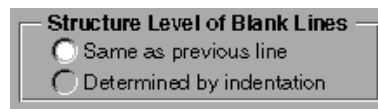


Figure 4-13 Structure Level of Blank Lines in Text Options

Alignment Options

Figure 4-14 on page 4-10 shows the alignment options for text. In the Indent field, enter the number of characters you want to shift right or left with the Text menu's Nest and Unnest commands. In the Tabs field, enter the number of characters you want between tab stops.

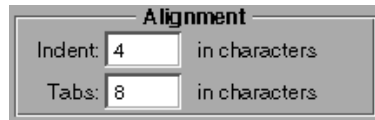


Figure 4-14 Alignment Options

Open at Structure Level Options

The options shown in Figure 4-15 determine how many levels of structure will be visible in a newly opened files. In the ASCII and RTF fields, enter a number between 0 and 99 to specify how many levels will be visible in a file of that type. A 0 indicates that only the top level of text, text that is flush left, will be visible; a 1 indicates that the first sublevel of text should also be visible, and so on.

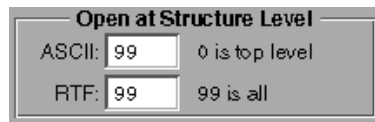


Figure 4-15 Open at Structure Level Options

Editing Modes

In addition to the default Text mode, there are two editing modes for C and Lisp source files, shown in Figure 4-16. These modes optimize some minor aspects of Edit's behavior for use with each of these programming languages. You can specify in the Modes field any additional file extensions that you want associated with either of these two modes.

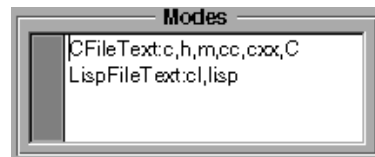


Figure 4-16 Editing Modes File Extensions

C Options

Choose C Options in the Preferences panel's pop-up list to see the C source code options that you can specify. C options are saved in your defaults database and continue to be used until you specify different values for them.

Structure for Top Level

Figure 4-17 shows the options that determine how the commands in the Structure submenu operate on top-level text. When you select the Independent of 1st character option, commands in the Structure submenu operate solely on the basis of indentation, independent of particular characters. Click on Determined by 1st character if you want Structure submenu commands to treat C preprocessor directives (lines whose first character is #) specially—that is, as second-level text, rather than top-level text.

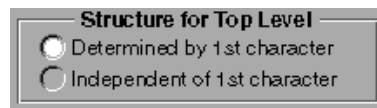


Figure 4-17 Structure for Top Level Options

Structure Level of Blank Lines

Figure 4-18 determines how blank lines are treated in your C source code structure. When you select the Same as previous line option, Edit assigns each “blank” line (that is, each line that contains no visible text) the same structure level as the previous line. Click on Determined by indentation if you want the structure level of blank lines to be determined by the amount of indentation (that is, tabs and spaces) on that line, rather than by the indentation of the previous line.

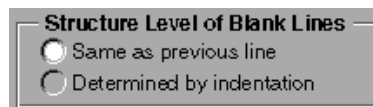


Figure 4-18 Structure of Blank Lines in C Code Options

Tags Path

Figure 4-19 shows the field that lets you specify one or more `tags` files that you want Edit to search when you choose the Source command in the Utilities menu. In the Path field, enter the path names of the files you want searched. A `tags` file, which you create using the UNIX `ctags` command, contains the locations of program object definitions among a given group of files. The Source command searches the `tags` files specified here for the location of an object definition and then opens the file containing the definition.

If you leave the default entry of `tags: ../tags` in this field, Edit searches only the main window) and in the current folder's parent folder. You can replace or add to the default, however, by entering the path names of one or more other `tags` files; you separate multiple path names with a colon as in a standard UNIX path list.

See the description of the Source command in “Commands in the Utilities Menu” on page 4-31 for more information about using Edit's Source command with `tags` files.



Figure 4-19 Tags Path

Include Path

Figure 4-20 on page 4-13 shows the field in which you can specify your default include path (the path the preprocessor uses to search for system header files). You can redefine this path by editing the text and then clicking on the Set button.

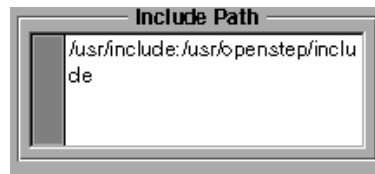


Figure 4-20 Include Path

Performing Basic Operations

For information about basic operations of Edit that are available in both User mode and Developer mode, see the following sections in *Using the OpenStep Desktop*:

- Selecting Text
- Deleting and Replacing Text
- Finding Text
- Replacing Text that You Find
- Checking Your Spelling
- Setting Margins, Indentation, and Tabs
- Checking Your Spelling

For more information about basic operations common to Edit and other standard OpenStep applications, see *Using the OpenStep Desktop*.

Opening Edit Files

In addition to opening Edit files from the workspace, you can open them from within Edit by using the Open or Open Selection commands in the File menu. (These commands are described in “Commands in the File Menu” on page 4-25.)

An alternate way to open one or more files is to use Edit’s `openfile` command at the UNIX prompt in a shell window. You can specify one or more file names (or path names), which are interpreted relative to the shell

window's current folder. For example, the following command would open all the files in the current folder that end with a `.c` extension, plus all the files in a subfolder called `headers` that end with a `.h` extension:

```
openfile *.c headers/*.h
```

Each file is opened in its own Edit window. You can use the `openfile` command only when Edit is running.

Using File Windows and Folder Windows

Edit provides two types of standard windows: *file windows* and *folder windows*. As in other applications, there are also panels and menus.

Note – Unless otherwise specified, folder windows mentioned in this chapter are Edit folder windows, not Workspace Manager folder windows.

An Edit file window displays a document file that you can view and edit. When you make changes to text displayed in a file window, the version of the file on the disk is not affected until you save the file with the File menu's *Save* command. When a file contains unsaved changes, the window's title bar displays a partially drawn close button. If you miniaturize a window containing unsaved changes, its miniwindow is highlighted in gray.

An Edit folder window displays a list of the files and subdirectories contained in a folder. You do not edit the contents of a folder window; instead, you use the displayed folder listing to find and select other files or directories to open.

Two special features are available in Edit folder windows:

- You can type a character to find and select the first item starting with that character. Each additional character you type deselects the previously selected item and finds the first item starting with the newly typed character. You can also use the commands in the Find submenu to find and select items in a folder window.
- You can double-click a file or folder name to open an Edit window displaying that file or folder. This is equivalent to selecting the name and choosing the Open Selection command in the File menu.

You can also open an Edit folder window by choosing the Open Folder command in the File menu. The command displays a panel in which you enter the path name of a folder to be opened.

Contracting and Expanding Text in a File Window

In Developer Mode, Edit provides a Structure capability that lets you quickly move around in C files as well as in any other type of file where levels of structure are represented by varying degrees of indentation—outlines, for example. Commands in the Structure menu can be used to “contract” text in the main window, displaying only the text at a particular level of indentation. Text that is indented beyond that level is hidden. Figure 4-21 shows a document that has been contracted—only the top-level lines (those that are flush left) are visible. Notice the two white text arrows, which indicate the presence of contracted text.

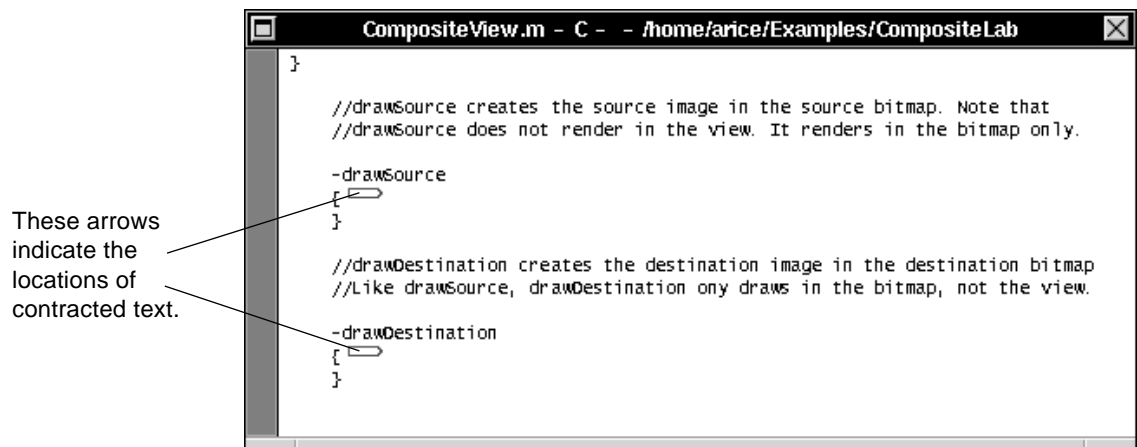


Figure 4-21 File Window with Only First-Level Text Expanded

When text is contracted, only the display is changed—the document itself, including font changes and text properties, remains unchanged. However, while some Edit commands affect both the expanded and the contracted portions of the document (for example, Cut and Paste), other commands affect only the portions of the document that are expanded (for example, commands that change the font).

Commands in the Structure submenu let you expand or contract either the entire contents of the window, or just the current selection. The rest of this section describes some mouse shortcuts that you will probably use even more frequently than the menu commands.

Clicking on a text arrow expands (that is, displays) the text that the arrow represents. Control-clicking on a text arrow expands just the top level of the text that the arrow represents. For example, Figure 4-22 shows what the `drawSource` definition looks like after Control-clicking on the first of the two text arrows shown in Figure 4-21. Notice that the `drawSource` definition has expanded, but the `drawDestination` definition is still contracted. Also notice that the `drawSource` definition has not expanded completely—the `switch` statement contains yet another level of contracted text.

```

}

//drawSource creates the source image in the source bitmap. Note that
//drawSource does not render in the view. It renders in the bitmap only.

-drawSource
{
    [source lockFocus];
    PScompositerect(0.0,0.0,sRect.size.width, sRect.size.height, NS_CLEAR);
    PSsetgray(sourceGray);
    PSsetalpha(sourceAlpha);
    PSnewpath();
    switch(sourcePicture) {
        [CIRCLE]
    }
    PSclosepath();
    PSfill();
    [source unlockFocus];

    return self;
}

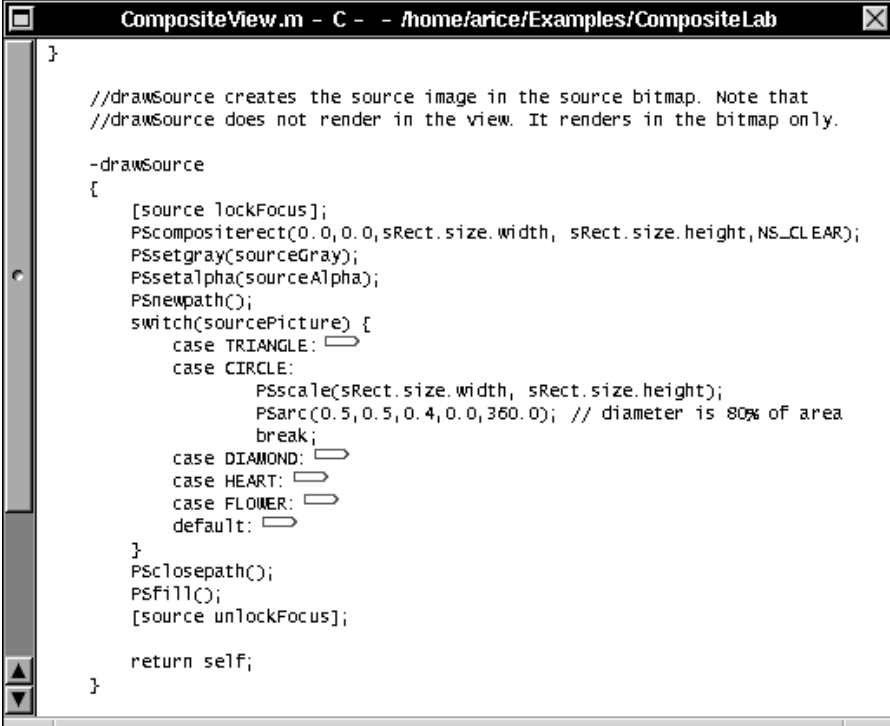
//drawDestination creates the destination image in the destination bitmap
//Like drawSource, drawDestination only draws in the bitmap, not the view.

-drawDestination
{
}

```

Figure 4-22 File Window with Some Second-Level Text Expanded

Figure 4-23 on page 4-17 shows the `drawSource` definition after Control-clicking on the `switch` statement’s text arrow. Each case statement in the `switch` contains an additional level of contracted text. The text for `CIRCLE`, however, is not contracted—it has already been expanded by clicking (or Control-clicking) on its text arrow.








```
}  
  
//drawSource creates the source image in the source bitmap. Note that  
//drawSource does not render in the view. It renders in the bitmap only.  
  
-drawSource  
{  
    [source lockFocus];  
    PScompositerect(0.0,0.0,sRect.size.width, sRect.size.height,NS_CLEAR);  
    PSsetgray(sourceGray);  
    PSsetalpha(sourceAlpha);  
    PSnewpath();  
    switch(sourcePicture) {  
        case TRIANGLE:   
        case CIRCLE:  
            PSScale(sRect.size.width, sRect.size.height);  
            PSArc(0.5,0.5,0.4,0.0,360.0); // diameter is 80% of area  
            break;  
        case DIAMOND:   
        case HEART:   
        case FLOWER:   
        default:   
    }  
    PSclosepath();  
    PSfill();  
    [source unlockFocus];  
  
    return self;  
}
```

Figure 4-23 File Window with Some Third-Level Text Expanded

If you want to recursively expand all the sublevels of text represented by a text arrow, click on, instead of Control-clicking on, the arrow.

Control-clicking anywhere within an indented block of text contracts the text.

Adding Help Links

The Help menu in Edit provides commands that are used to add or edit Help links. Although Help links are designed for use within an application's on-line Help system, they can also be used more generally. For example, the Contents file for the release notes could contain links to the various release note files. For information about adding a Help system to an application you are developing, see "Commands in the Project Menu" on page 2-33 and "Attaching Help to Objects" on page 3-132.

To work with Help links and markers, use the following commands in the Help menu (choose the `Help` command in the Format menu):

- Choose **Insert Link** to insert a Help link at the current insertion point. In the Link Inspector that appears, specify the name of a file and (optionally) a marker in that file.
- Choose **Insert Marker** to insert a Help marker at the insertion point in the main window. A Marker panel appears in which you specify a name to associate with the marker. When you insert a link to the marker, you will identify it by this name.
- Choose **Show Markers** to show all the markers in the main window, or **Hide Markers** to hide them.

If you want to edit a link or marker you have created, Command-click on it to bring up the Inspector panel. To delete a link or marker, select it and press the Delete key, just as you would with text.

Using Templates

Three commands on the Expert menu—Expansion Dictionary, Insert Field, and Next Field—let you create and use text string abbreviations. Text string abbreviations are abbreviations for commonly used text strings or templates that you can type and then expand into the full text entry with a single keystroke.

To define a text string abbreviation, open the Expansion Dictionary panel by choosing Expansion Dictionary in the Expert menu.

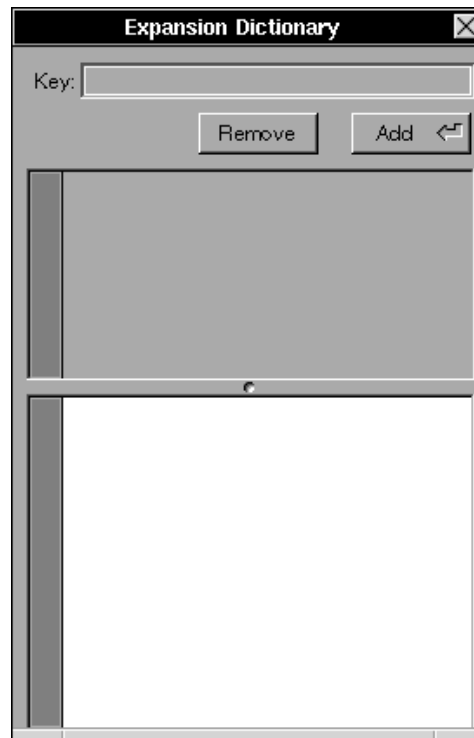


Figure 4-24 Expansion Dictionary Panel

In the Key field at the top of the panel, enter an abbreviation for the text string or template. In the Expansion field at the bottom of the panel, enter the expanded text that you want the abbreviation to represent. Then click on the Add button. The new abbreviation appears on the list in the middle of the panel.

If you want the expansion to occupy more than one line, press Alt-Return to insert Return characters between lines in the Expansion field. When you press Alt-Return, the line of expanded text you just typed disappears from the field, leaving room to type the next line.

To use a text abbreviation, set the Key Bindings On option in the Emacs Key Bindings field of Global Options in the Preferences panel (see “Global Options” on page 4-6), type the abbreviation in a document, and then press the Escape key; the abbreviation is replaced by its expansion. For example, if you

frequently need to type `setOutputForm`, you could use the Expansion Dictionary command to associate the abbreviation `sof` with the longer declaration. To enter `setOutputForm`, you would only have to type `sof` and press Escape. The abbreviation does not even have to be typed in full for the expansion to occur, as long as what you do type refers unambiguously to a glossary entry.

If you are using the Expansion Dictionary window to create a template containing fields you will be editing after the text is expanded, surround each field with European quotation marks (`«»`), as described below. For example:

```
Subject: «subject»  
To: «recipient»  
cc: «cc»»
```

```
«message»
```

You can enter European quotation marks in the Expansion field by choosing the Insert Field command, or you can enter them directly from the keyboard by pressing and releasing the Compose key and then typing `Shift-<-<`, then pressing and releasing the Compose key and typing `Shift->->`. After inserting the template into a document, you can quickly find each editable field by choosing the Next Field command, which positions the insertion point at the next field in the template.

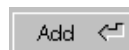


Figure 4-25 Add Button

After entering the abbreviation and the expanded text it stands for in the Key and Expansion fields, click on the Add button to accept the new glossary entry.

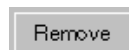


Figure 4-26 Remove Button

To remove a glossary entry, type its abbreviation in the Key field and click on the Remove button.

Using Keyboard Editing Commands

In addition to letting you edit text using menu commands and their keyboard equivalents, Edit also supports several Emacs-style editing commands that can be typed from the keyboard. Table 4-2 lists the key combination corresponding to each of these commands and a description of what the command does.

Table 4-2 Keyboard Editing Commands

Command	Action
Control-B	Moves back one character
Control-F	Moves forward one character
Alt-b	Moves back one word
Alt-f	Moves forward one word
Control-A	Moves to beginning of line
Control-E	Moves to end of line
Control-D	Deletes next character
Control-H	Deletes previous character
Alt-d	Deletes to end of current (or next) word
Alt-h	Deletes to beginning of current (or previous) word
Control-K	Deletes forward to end of line
Alt-<	Moves to beginning of text
Alt->	Moves to end of text
Control-N	Moves down one line
Control-P	Moves up one line

Interacting with UNIX

Edit provides some useful commands for using UNIX utilities from within Edit. These include the following:

- Two commands for piping output from UNIX commands directly into Edit files
- A `source` command that you can use with one or more `tags` files to locate program objects in a group of files

Piping UNIX Output to a File

Edit lets you pipe the output of a UNIX command directly into an Edit window. This is a useful technique for inserting output from other applications into your own programs.

For example, to produce a 1996 calendar in an empty window, choose Command in the Utilities menu, enter the following in the panel that appears, and press Return:

```
cal 1996
```

The output appears in an untitled window.

If instead you wanted the calendar to appear in the main window, position the insertion point where you want the calendar to appear (or select what you want it to replace). Then choose Pipe in the Utilities menu. Enter the same command as before and press Return. This time the output appears in the main window at the insertion point or in place of the current selection.

You can also use the Pipe command to manipulate the current text selection with another UNIX program. If the command accepts input, the selection will be used as input—for example, you could sort the selection with the `sort` command.

If there are Command and Pipe commands that you use frequently, you can define them as menu items in the User Commands and User Pipes submenus in the Utilities menu. To do this, enter a definition for each command in a file named `.openstep/.commanddict` or `.openstep/.pipedict` in your home folder.

Each command definition contains at least two fields, separated by tabs:

```
command name<tab>command definition
```

For example, the following entry defines a Pipe command called `Sort Selection`, which runs the UNIX `sort` command using the current selection as input:

```
Sort Selection    sort -
```

One additional field, inserted between the two required fields and separated from them by tabs, can be used to specify a keyboard alternative for the command. For example, this definition of the `Sort Selection` command assigns to it the keyboard alternative `Command-5`:


```
Sort Selection      5      sort -
```

If you make changes to your `.commanddict` or `.pipedict` file while Edit is running, you must quit and restart Edit in order for your changes to appear in the User Commands or User Pipes menu.

You can use two special variables, shown in Table 4-3, as arguments to the UNIX commands you specify:

Table 4-3 Arguments for UNIX Commands

Argument	Meaning
<code>\$file</code>	Refers to the file that is displayed in the main window (which may be different from the contents of the window).
<code>\$selection</code>	Refers to the contents of the current selection, which can be either text that is selected in a file window, or a file that is selected in a folder window

Here are some examples of how these variables might be used in a `.commanddict` definition:

```
Print Two Up      P      enscript -2r $file
GrepAppkit       A      fgrep -n "$selection" /usr/include/appkit/*.h
```

The first example prints the contents of the file that is displayed in the main window. The second example searches for occurrences of the selected text in the Application Kit header files.

Using a Tags File

If you are maintaining a large number of files as part of a programming project, you can use Edit's Source command with a `tags` file to quickly locate the definition of an object in that group of files. A `tags` file (which you create with the UNIX `ctags` command) lists the locations of program objects (such as functions, procedures, global variables, and typedefs) that are in a specified group of files.

To locate an object definition, simply select it and choose Source (or choose Source and type the object name in the panel that appears). Edit searches one or more `tags` files for the location of the object definition and then opens the file containing the definition. Normally, Edit searches the `tags` file in the current folder (the folder containing the file in the main window). However,

you can specify other `tags` files to be searched either with the Preferences command or by specifying the `Tags` option when starting up Edit from a shell window.

More information on `tags` files is given in the `ctags` UNIX manual page. For more information on using the `Source` command, see the command description in “Commands in the Utilities Menu” on page 4-31.

Edit Command Reference

The following sections summarize the menus and commands available in Edit.

Commands in the Main Menu

Edit’s main menu contains the standard Info, Print, Windows, Services, Hide, and Quit commands. The other commands and the submenus they open are described in the sections that follow. Several standard commands are discussed here only in terms of their particular use in Edit.

Commands in the File Menu

Edit's File menu contains the standard Open, New, Revert to Saved, and Close commands. The other commands are described in Table 4-4.

Table 4-4 File Menu Commands

Command	Description
Save, Save As, Save To, Save All	<p>These are the standard commands for saving the contents of the main window on the disk.</p> <p>When you save a file, Edit first moves the contents of the old version to a temporary backup file, which has the same name as the previous file but with a tilde (~) appended to it (for example, the backup file corresponding to <code>Fruit.m</code> would be <code>Fruit.m~</code>).</p> <p>Next, Edit writes the new version of the file and then it normally deletes the backup file. If something happens that prevents Edit from saving the file, however, the backup file remains so you can recover its contents. Or, if you always want the backup file to remain, even after the new version is successfully saved, you can set the Don't delete backup file option in the Preferences panel.</p> <p>While the file is being saved, <code>saving:</code> appears before the file name in the title bar of the window (in the case of small files, it appears only for an instant). Until <code>saving:</code> has disappeared, do not use the file (for example, do not try to compile or copy it).</p>
Open Selection	<p>Opens the file or folder currently selected in the main window. Normally, you use this command on a selection in a folder window. However, it also works on selected text in a file window. The selected text must be either a full path name, or a file name or path name relative to the current folder (the folder containing the file in the main window).</p>
Open Folder	<p>Displays a panel in which you enter the path name of a folder to be opened. When you click on OK, the folder opens in an Edit folder window. When the panel appears, Edit displays the name of the current folder in the Folder name field.</p>

Commands in the Edit Menu

Edit's Edit menu contains the standard Cut, Copy, Paste, Delete, and Select All commands, plus commands for opening the Link submenu and the Find submenu described in "Commands in the Link Submenu" and "Commands in the Find Submenu." Other commands are described in Table 4-5.

Table 4-5 Edit Menu Commands

Command	Description
Undelete	Reinserts the most recently deleted text, even if the text has not been put on the pasteboard. You can insert the deleted text at a new location by positioning the insertion point where you want to insert the text (or selecting text that you want it to replace) and then choosing Undelete.
Spelling	Opens the Spelling Panel for checking the spelling of words in the main window. See "Checking Your Spelling" in <i>Using the OpenStep Desktop</i> .
Check Spelling	Has the same effect as clicking Find Next in the Spelling panel—that is, it finds the next word not contained in the spelling dictionary. See "Checking Your Spelling" in <i>Using the OpenStep Desktop</i> .

Commands in the Link Submenu

Edit's Link submenu provides the commands described in Table 4-6 for working with linked documents. For more information, see "Adding Linked Graphics" in *Using the OpenStep Desktop*.

Table 4-6 Link Submenu Commands

Command	Description
Paste and Link	Pastes a copy of a graphic contained on the pasteboard, but creates a link to the document that the graphic came from, so that future changes to the original graphic will affect the copy in the Edit document as well.
Show Links, Hide Links	Shows (or hides) whether or not graphics are linked by displaying a linked chain around the border of each linked graphic.
Link Inspector	Opens the Link Inspector panel.

Commands in the Find Submenu

Edit's Find submenu contains the standard Find Panel, Find Next, Find Previous, and Enter Selection commands. Other commands are described in Table 4-7.

Table 4-7 Find Submenu Commands

Command	Description
Jump to Selection	Scrolls the insertion point or current text selection into view.
Line Range	<p>Opens a panel that identifies by number the line or line range containing the current selection in the main window. If the Character option in this panel is selected instead of the Line option, then the character range is displayed instead of the line range.</p> <p>You can also use the panel to search for a particular line, line range, character, or character range in the main window. Enter a number or a range (a range is two numbers separated by a colon) in the Range field. Click on the Select button to select that character, line, or range of the file.</p>

Commands in the Format Menu

The Format menu contains commands for displaying the standard Font and Text submenus, as well as Edit-specific Help and Structure submenus (see Table 4-9 on page 4-29 and Table 4-10 on page 4-30). It also contains the Page Layout command described in Table 4-8.

Table 4-8 Format Menu Commands

Command	Description
Page Layout	<p>Displays the standard Page Layout panel for choosing among various paper sizes, scaling factors, and orientations for text printed from the main window.</p> <p>When you print text that is displayed in a window, the printed words wrap exactly as they are wrapped on the screen. Therefore, if you change the page layout, the width of the window may also need to be changed in order for the text to print correctly. Changing the page layout does not affect the size of the main window, so you need to make this adjustment.</p>

Commands in the Font Submenu

The Font submenu contains the standard Font commands, plus a few additional commands that let you change the font properties of the text displayed in the main window—for example, the Colors command opens the standard Colors panel, which you can use to change the color of the selected text.

In an RTF file, font changes apply to the current selection and are saved when you save the contents of the window.

In an ASCII file, font changes are applied to the entire contents of the main window—font changes in non-RTF files are not saved when you save the contents of the window.

Commands in the Text Submenu

Edit's Text submenu contains commands (see Table 4-9) that let you change properties of the text displayed in the main window. Some of these commands work only on text in RTF files; use the Make Rich Text command if you want to change the contents of the main window from ASCII to RTF.

Table 4-9 Text Submenu Commands

Command	Description
Align Left, Center, Align Right	These align the text with the left margin (ragged right), center it between both margins, or align it with the right margin (ragged left).
Make Rich Text, Make ASCII	Changes the format of the text in the main window from RTF to ASCII, or vice versa. In an RTF file, font changes and other text properties (such as superscripting and subscripting) can be saved as part of the file and displayed along with the text.
Nest, Unnest	These commands help you indent blocks of program code. Select the program lines you want to indent and then choose Nest. Each line in the selected program text will be indented the default amount (four characters, unless you have specified a different default value in the Preferences panel or overridden the default when you started up Edit from a shell window). Unnest moves the selected lines the default number of characters to the left, thus counteracting the effect of Nest.
Show Ruler, Hide Ruler	Show Ruler displays a ruler at the top of the main window, and the Hide Ruler command removes it. With this ruler you can set margins, tabs, and paragraph indentation. See "Using the Ruler" in <i>Using the OpenStep Desktop</i> .

Table 4-9 Text Submenu Commands (Continued)

Command	Description
Copy Ruler, Paste Ruler	Copy Ruler copies the ruler settings of the paragraph containing the insertion point or the beginning of the current selection, so that you can subsequently paste them with Paste Ruler. It is as though there's a separate pasteboard for the ruler, and Copy Ruler replaces what is already on it, just as Copy does for text.
	Paste Ruler affects the paragraph containing the insertion point or the current selection. It replaces the paragraph's ruler settings with the last ones you copied with Copy Ruler. If the current selection spans more than one paragraph, Paste Ruler replaces the ruler settings of all the selected paragraphs.
	These commands do not require the ruler to be showing, and they do not change the contents of the pasteboard.

Commands in the Help Submenu

The Help submenu provides the commands described in Table 4-10, which are used to add or edit Help links. Although Help links are designed for use within an application's on-line Help system, they can also be used more generally (for example, the Contents file for the release notes could contain links to the various release note files). For more information about working with Help links and markers, see "Adding Help Links" on page 4-17. For information about adding a Help system to an application you are developing, see "Commands in the Project Menu" on page 2-33 and "Attaching Help to Objects" on page 3-132.

Table 4-10 Help Submenu Commands

Command	Description
Insert Link	Inserts a Help link at the insertion point in the main window.
Insert Marker	Inserts a Help marker at the insertion point in the main window.
Show Markers, Hide Markers	Shows (or hides) all the markers in the main window.

Commands in the Structure Submenu

The Structure submenu provides commands (see Table 4-11) that control whether certain portions of the text in the main window are expanded (that is, visible) or contracted (that is, hidden). These commands are useful for working with files that have a regular multilevel structure, in which the various levels of structure are represented by varying degrees of indentation; for example, an outline or Objective C language source code. See “Contracting and Expanding Text in a File Window” on page 4-15 for a detailed introduction to this Edit feature.

Table 4-11 Structure Submenu Commands

Command	Description
Contract All, Expand All	These contract or expand all the text in the main window.
Contract Sel, Expand Sel	These contract or expand the selected text in the main window.

Commands in the Utilities Menu

Commands in the Utilities menu, described in Table 4-12 on page 4-32, perform a variety of functions such as providing an interface to the UNIX shell and looking up information in a UNIX manual page. There are also two customizable submenus—User Commands and User Pipes—to which you can add commands that you have defined yourself.

Table 4-12 Utilities Menu Commands

Command	Description
Command	<p>Displays a panel in which you specify a UNIX command to be run. The output of the command appears in a window titled UNTITLED, rather than in the main window.</p> <p>Two variables can be used as arguments to the UNIX command you specify:</p> <p><code>\$file</code> refers to the file that is displayed in the main window.</p> <p><code>selection</code> refers to the contents of the current selection, which must be single file specification (wildcards can be used). Normally this will be a file that is selected in a folder window.</p>
User Commands	<p>Displays a submenu of commands you have defined and saved in a file named <code>.openstep/.commanddict</code> in your home folder. Any changes you make to the <code>.commanddict</code> file do not take effect until the next time you start Edit. The <code>.commanddict</code> file format is described in “Piping UNIX Output to a File” on page 4-22.</p>
Pipe	<p>Works the same as Command, with one important difference: The output of the UNIX command that you specify is not displayed in another window—instead, the output (including any error messages that might be generated) appears in the main window at the insertion point or in place of the current selection.</p>
User Pipes	<p>Displays a submenu that contains pipe commands you have defined and saved in a file named <code>.pipedict</code> in your home folder. These commands may be similar to commands you define in the User Commands menu, but the output appears in the main window at the insertion point or in place of the current selection, rather than in a separate window.</p> <p>The <code>.pipedict</code> file format is described earlier in “Piping UNIX Output to a File” on page 4-22.”</p>

Table 4-12 Utilities Menu Commands (Continued)

Command	Description
Source	<p>Opens the file containing the definition of the program object (such as a function, procedure, global variable, or typedef) selected in the main window. This command searches one or more <code>tags</code> files for the location of the object definition and then opens the file containing the definition. Normally, Edit searches the <code>tags</code> file in the current folder (the folder containing the file in the main window). However, you can specify other <code>tags</code> files to be searched either in the Preferences panel or when starting up Edit from a shell window.</p> <p>To locate an object definition, select the function name, macro, or other program object in the file you are working in and choose Source. Edit opens the file containing the required information and highlights the first occurrence of the object in the text. If you choose Source without selecting text, Edit displays a panel that prompts you to enter the program object you want defined. If Edit cannot locate the object, it informs you that no such <code>tags</code> file entry for the object exists. (If this happens, use the Preferences command to make sure that the path name of the <code>tags</code> file listing the location of the object is specified.)</p> <p>A <code>tags</code> file is a file you create with the UNIX <code>ctags</code> command. The file lists the locations of specified program objects (such as functions, procedures, global variables, and typedefs). More information on <code>tags</code> files is given in the <code>ctags</code> UNIX manual page.</p>
Manual	<p>Displays a UNIX manual page in an Edit window. First select the manual page subject in the main window and then choose Manual. If there is no selection, a panel appears prompting you for an entry.</p>
Match	<p>If you select one of a matching pair of delimiters (parentheses, braces, or square brackets) and choose Match, the pair of delimiters and the enclosed text become selected. You can also invoke this command by double-clicking on either of the delimiters.</p>

Commands in the Expert Submenu

The Expert menu provides the advanced commands described in Table 4-13.

Table 4-13 Expert Menu Commands

Command	Description
Update Folder	Updates the contents of the main window, which must be a folder window. Folder windows are not automatically updated, so this command is useful when files in a folder have been created, deleted, or renamed.
Copy PS	Copies the contents of the main window onto the pasteboard as an Encapsulated PostScript (EPS) image. Once pasted into an application that accepts EPS images, the pasted copy of the text can no longer be edited.
Expansion Dictionary	Opens the Expansion Dictionary panel for managing text expansion definitions. See “Using Templates” on page 4-18 for a complete description of this panel.
Insert Field	Creates a new field in an expansion template. See “Using Templates” on page 4-18.
Next Field	Moves the insertion point to the next field in an expansion template. See “Using Templates” on page 4-18.
Close Ancestors	Closes all Edit windows associated with each folder that is neither the main window’s folder nor one of its subfolders.
Close Descendants	Closes all Edit windows associated with each folder that is a subfolder of the main window’s folder. If the main window is a folder window it will remain open, but if the main window is a file window it will be closed as well.

Using Icon Builder to Create Application Icons



The Icon Builder application is a simple yet effective tool—either alone or in combination with a more powerful drawing application—for creating application icons. Although Icon Builder itself is not intended to be a full-featured drawing application, it offers not only integration with other drawing applications, but also the ability to create and edit multiple-icon documents.

You can start Icon Builder, which is located in `/usr/openstep/Developer/Apps`, from the workspace as you would any other application—by double-clicking its icon in the workspace. When Icon Builder starts up, it displays a panel of tools used to edit icon documents.

Creating, Opening, and Saving Documents

When Icon Builder starts, it creates one new Icon Builder window using the default Preferences settings. You can create additional Icon Builder windows as you need them, as described in “Creating a New Document” on page 5-2.

Creating a New Document

To create a new document, choose the New command in the Document menu. This creates a document with the default attributes. Typically, the document contains a single 48-pixel by 48-pixel, non-alpha, 2-bit gray image with a white background. You can change the attributes of the document after it's been created, as described in “Changing the Attributes of a Document” on page 5-17.

To create a new document with nondefault attributes, do the following:

1. **Bring up the New Document panel shown in Figure 5-1 by choosing the New Layout command in the Document menu.**

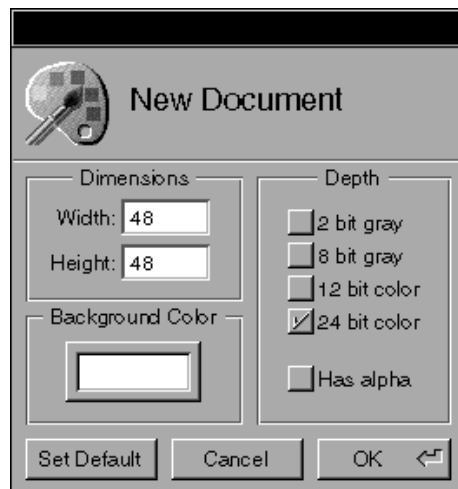


Figure 5-1 New Document Panel

2. **Set options in the New Document panel, and then click on OK to create a new document. If you want to change the default attributes for all documents created with the New command, click on Set Default instead.**

For example, if you want to create an icon for use on both color and black and white displays, you would check the 2 bit gray box as well as the 12 bit color box (8 bit gray and 24 bit color could also be used). Check the Has alpha box if you will be using alpha. (Alpha means transparency, so that some of the

background color shows through an image.) To change the background color, choose Colors on the Tools menu to bring up the Colors panel, pick a color in the Colors panel, and drag the color into the Background Color color well.

Opening an Existing Document

To open an existing document, choose the Open command in the Document menu and use the Open panel to find the document.

The document you open may be an icon you are working on, or it may simply contain an image that you want to copy a selection from in order to paste it into another document. In addition to TIFF files, Icon Builder can open GIF and EPS files.

Saving a Document

To save a document to a file, choose the Save command in the Document menu. If the document has not been saved yet, a Save panel is displayed prompting you to specify a name and location for the file.

Even if the file you are saving was opened as something other than a TIFF file (a GIF file, for instance), it will be saved as a TIFF file.

Icon Builder saves TIFF files in uncompressed format, so before making the file part of your application project, you should use the `tiffutil` utility to compress the file. See the `tiffutil(1)` UNIX manual page for more information.

Editing Icon Documents

This section describes various ways to edit an icon document, including the set of Icon Builder tools and an inspector for fine-tuning those tools. Other editing techniques involve zooming in and out, changing the attributes of a document, and working with multiple-icon documents.

You can also use standard cut, copy, and paste techniques; these are not described here.

Using Icon Builder Tools

A variety of drawing tools are available and accessible from the Tools panel, shown in Figure 5-2, which is displayed automatically when you start Icon Builder. If you close or misplace the panel, you can retrieve it by choosing the Tools command in the Tools menu.



Figure 5-2 Tools Panel

To use a tool, select it by clicking on its icon in the Tools panel. Once you have selected a tool, use it to edit the contents of the document window.

Note – When using the Tools panel, you should have the Colors panel open as well. All the drawing tools use the color that is currently displayed in the Colors panel. You can also use the Colors panel to specify the degree of alpha coverage (that is, opacity), as well as whether or not painting is done in overlay mode.

The Brush Tool

The Brush tool, shown in Figure 5-3, is useful for filling in large areas with a particular color. Click once to deposit a brushful of color, or click and drag to cover a larger surface area.



Figure 5-3 The Brush Tool

The Line Tool

The Line tool, shown in Figure 5-4, draws straight lines. Click to mark the start point, and drag to the endpoint. The line you see being drawn as you drag is only for guidance—the final line is drawn only when you release the button.



Figure 5-4 The Line Tool

The Oval Tool

The Oval tool, shown in Figure 5-5, draws circles and ovals. Click and drag to determine the position, size, and shape. It is hard to predict the startpoint and endpoint with accuracy, so you may want to use another document window as a scratch area and then copy and paste the oval once you are satisfied with it.



Figure 5-5 The Oval Tool

The PaintBucket Tool

The PaintBucket tool, shown in Figure 5-6, changes the color of a contiguous, identically colored group of pixels. The color to which they are changed is the color that is currently in the Colors panel. Before using the Paint Bucket tool, you may want to use the ObeseBits panel to be sure that all the pixels are in fact identical in color—if minor gradations in color are used to achieve the appearance of a particular shade, the new color will not spread from pixel to pixel.



Figure 5-6 The PaintBucket Tool

The Pencil Tool

The Pencil tool, shown in Figure 5-7, draws freehand lines. Click to start the line, and drag to indicate the path of the line. Unlike the line tool, the Pencil tool draws the final line as you drag. If you do not like the result, use the Undo command in the Edit menu to undo it.



Figure 5-7 The Pencil Tool

The Rectangle Tool

The Rectangle tool, shown in Figure 5-8, draws squares and rectangles. Click to position a corner point, and then drag in any direction to form the rectangle.



Figure 5-8 The Rectangle Tool

The Selection Tool

The Selection tool, shown in Figure 5-9, selects a rectangular area for further editing. For example, after selecting an area you might go on to copy the selection to the pasteboard, or even delete the selection. You can deselect the selection by clicking anywhere in the document window.



Figure 5-9 The Selection Tool

The Text Tool

The Text tool, shown in Figure 5-10, is used to add text to an image. If you select the Text tool and then click the cursor in a document window, the contents of the TextTool inspector (by default, the word Text—probably not what you want in your icon) are copied to the cursor location. As long as you do not release the mouse button you can drag the text to reposition it, but once you let go the text becomes fixed in that position.



Figure 5-10 The Text Tool

In order to use the Text tool effectively, you should first enter the desired text in the TextTool inspector. To make changes to the font of the text, choose Font on the Format menu, and then choose Font Panel from the Font menu. Use the font panel to set the font attributes and font size as you want. Then use the Text tool to insert the text in the document window, or—to be on the safe side—insert the text first in a temporary scratch document, and then cut and paste the text into the document window.

Other tools besides the Text tool have default attributes that you can change using the Tools Inspector, as described in “Using the Tools Inspector” on page 5-8.

Using the Tools Inspector

The Tools inspector is a panel that gives you greater control over the characteristics of the tools available in the Tools panel.

To bring up the Tools inspector, choose the Inspector command from the Tools menu. To display the inspector for a particular tool, click on the tool's icon in the Tools panel.

Note – There is no inspector available for the PaintBucket tool.

The Brush Inspector

The Brush inspector, shown in Figure 5-11, is displayed in the Tools inspector panel when you select the Brush tool in the Tools panel.

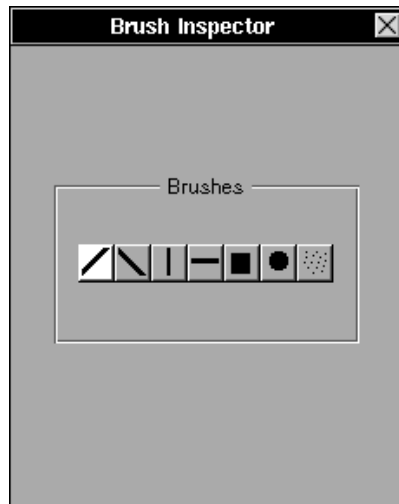


Figure 5-11 The Brush Inspector

Click on a button to change the shape and orientation of the brushstrokes you make.

The Line Inspector

The Line inspector, shown in Figure 5-12, is displayed in the Tools inspector panel when you select the Line tool in the Tools panel. You can use this inspector to change the width and end shape of the lines you draw.

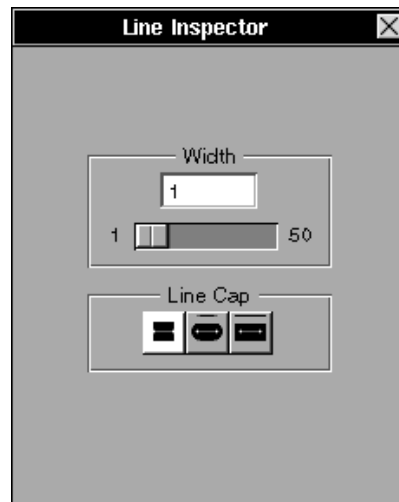


Figure 5-12 The Line Inspector

Use the slider or the text field to set the line width to any value between 1 and 50 pixels.

Click on one of the three Line Cap buttons to determine how the ends of lines are drawn. The setting becomes less critical as the line width decreases—a one-pixel line is drawn the same no matter what style of line cap is selected.

The Oval Inspector

The Oval inspector, shown in Figure 5-13 on page 5-10, is displayed in the Tools inspector panel when you select the Oval tool in the Tools panel.

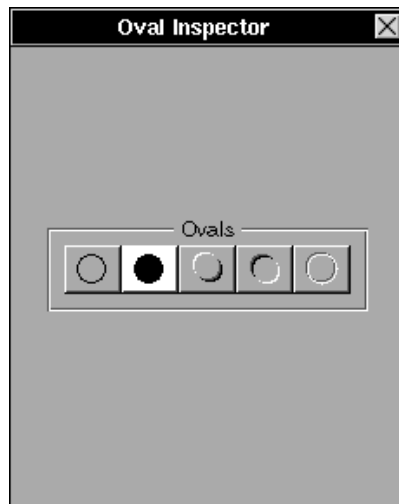


Figure 5-13 The Oval Inspector

Click on one of the five buttons to change the appearance of the circles and ovals you draw.

The Pencil Inspector

The Pencil inspector, shown in Figure 5-14 on page 5-11, is displayed in the Tools inspector panel when you select the Pencil tool in the Tools panel. You can use this inspector to change the width of the freehand lines you draw.

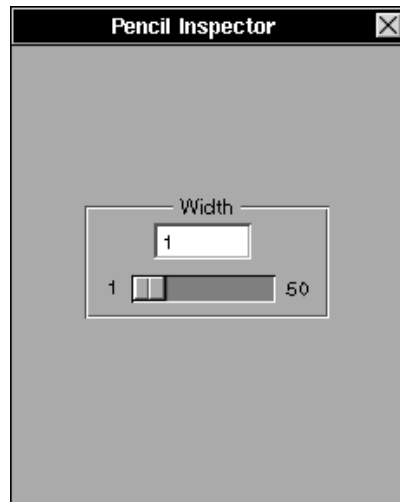


Figure 5-14 The Pencil Inspector

Use the slider or the text field to set the line width to any value between 1 and 50 pixels. Thicker lines cause the drawing speed to decrease, so you may need to move the mouse more slowly in order for the drawing process to keep up with it.

The Rectangle Inspector

The Rectangle inspector, shown in Figure 5-15 on page 5-12, displayed in the Tools inspector panel when you select the Rectangle tool in the Tools panel. You can use this inspector to change the appearance of the squares and rectangles you draw.

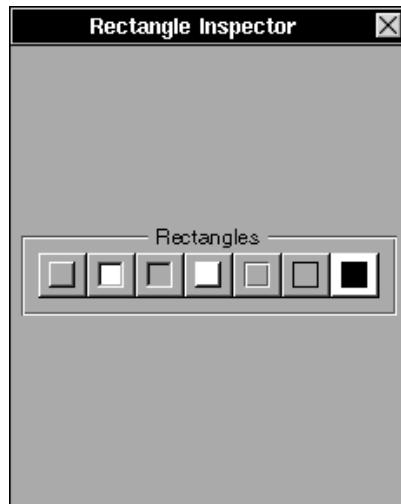


Figure 5-15 The Rectangle Inspector

Click on one of the seven buttons to change the appearance of the squares and rectangles you draw.

The Selection Inspector

The Selection inspector, shown in Figure 5-16 on page 5-13, is displayed in the Tools inspector panel when you select the Selection tool in the Tools panel. You can use this inspector to change the orientation of (that is, flip or rotate) the current selection.

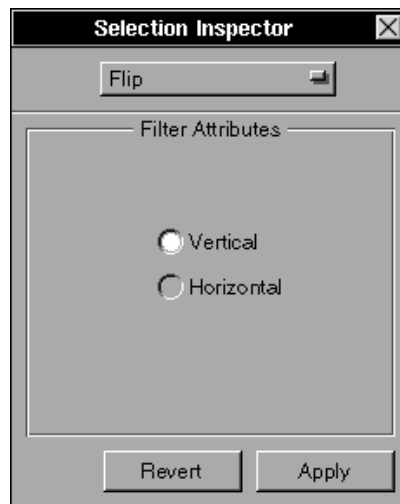


Figure 5-16 The Selection Inspector

Choose Flip or Rotate in the pop-up list at the top of the panel. The available options vary depending on which you choose.

If you choose Flip, you will see the Flip filter attributes shown in Figure 5-17. Click on either the Vertical or Horizontal button to indicate the direction in which you want the selection to be flipped.

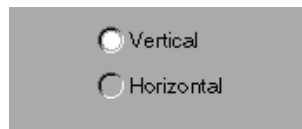


Figure 5-17 Flip Filter Attributes

If you choose Rotate, you will see the Rotate filter attributes shown in Figure 5-18 on page 5-14. Specify a value between 0 and 360 using either the slider or the text field. This value represents the number of degrees the selection will be flipped in a clockwise direction.

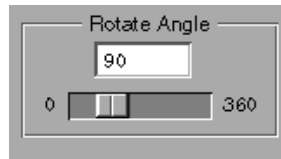


Figure 5-18 Rotate Filter Attributes

Once you have selected the options, click on one of the buttons shown in Figure 5-19. Click on Apply to flip or rotate the selection. If you do not like the results, click on Revert to return the selection to its former orientation.



Figure 5-19 Revert and Apply Buttons

The TextTool Inspector

The TextTool inspector, shown in Figure 5-20 on page 5-15, is displayed in the Tools inspector panel when you select the Text tool in the Tools panel. You use this inspector to input the text to be inserted in the document window, as well as to change the font attributes and formatting of the text prior to inserting it.

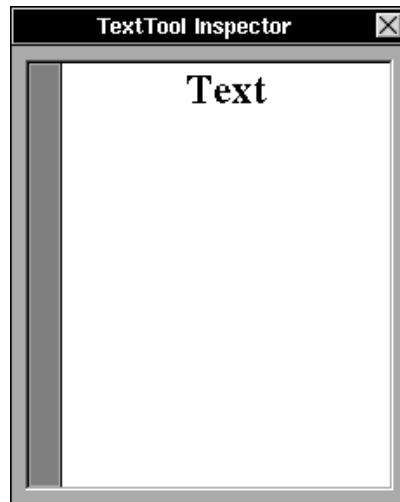


Figure 5-20 The TextTool Inspector

Type the text you want to insert in the document window. Use the Font panel to set the font attributes and size of the text. You may also want to use commands in the Text menu to format the text. Then select the Text tool and click in the document window to insert the text in the document.

Zooming In on a Document

When you are doing detail work on an image that is only 48 pixels across, you may find yourself wishing you had a magnifying glass. If you start feeling this way, choose the ObeseBits command in the Tools menu to bring up the ObeseBits panel shown in Figure 5-21 on page 5-16.

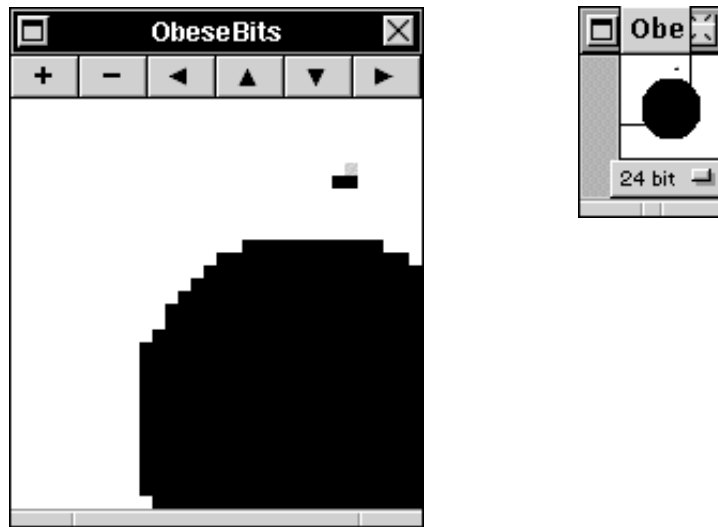


Figure 5-21 The ObeseBits Panel

This panel magnifies the image in the main window, and lets you zoom in or out (that is, increase or decrease the degree of magnification). The buttons along the top of the panel give you the control you need—use the plus and minus buttons to zoom in or out, and the arrow buttons to change the portion of the image being displayed.

There are actually two ObeseBits panels, as shown in Figure 5-21. The larger panel is for editing; the small panel sits over the document window and shows the exact portion of the image that is contained in the large panel. You can drag the small panel by its title bar, thereby changing the portion of the image being displayed in the panel.

The ObeseBits panel associates itself with whatever window is the main window. Clicking on the document containing the Mail icon in the figure, for example, would cause the small ObeseBits panel to jump to that document window. The contents of the big panel would change accordingly.

Note – Although the drawing tools in the Tools panel can be used directly in the ObeseBits panel, the result is not always intuitive. For example, the size of the Brush cursor does not accurately represent the brush size that’s used when stroking the brush. Use the Undo command in the Edit menu to undo any changes that you regret making.

Changing the Attributes of a Document

After you have created a document or opened an existing document, you may find it necessary to change its size, format, or other characteristics. For example, you might decide to add alpha (image transparency) to a document that does not have it.

To make such changes, first click on the document window to make sure it is the main window. Then open the Document Layout panel by choosing the Document Layout command in the Format menu.

The Document Layout panel is identical to the New Document layout panel (see Figure 5-1 on page 5-2). The only difference is that this panel is used to change the attributes of an existing document, rather than determine the attributes of a new document.

Note – If you set new default attributes in the Document Layout panel, these become the default attributes for the New Document layout panel as well.

Working with Multiple-Icon Documents

One Icon Builder document (that is, one TIFF file) can contain more than one icon. This is typically the case, for example, when you want to have one icon for color monitors and another for gray-scale monitors—if the two icons are in the same TIFF file, the appropriate icon is displayed automatically on each type of monitor, without any work on your part.

To create a multiple-icon document (or to change an existing single-icon document to a multiple-icon document), select the desired depth settings in the New Document panel (or the Document Layout panel). Then click on OK.

Use the pop-up list that is displayed in the lower right corner of a multiple-icon document window to access the various icons.

If you create a multiple-icon document, remember that you have to edit each icon. When you save the document, all the icons in the document are saved—not just the one that is currently visible in the document window.

Icon Builder Command Reference

This section describes the application-specific menus and commands available in Icon Builder. For descriptions of standard menus and commands, see *Using the OpenStep Desktop*.

Commands in the Main Menu

Icon Builder's main menu contains the standard Edit, Windows, Print, Services, Hide, and Quit commands. The Document, Format, and Tools commands display submenus that are described in Table 5-1 and the sections that follow.

Table 5-1 Icon Builder Submenus

Submenu	Description
Document	Displays a submenu of commands for creating, opening, and saving document windows. See "Commands in the Document Menu."
Format	Displays a submenu of commands for opening the standard Font and Text submenus, plus the Document Layout command for specifying the layout of the document in the main window. See "Commands in the Format Menu" on page 5-19.
Tools	Contains commands for opening a panel containing the tools available for use in creating an icon. See "Commands in the Tools Menu" on page 5-20.

Commands in the Document Menu

Icon Builder's Document menu provides the commands described in Table 5-2 on page 5-19.

Table 5-2 Icon Builder’s Document Submenu

Command	Description
Open	Opens an existing document window. See “Opening an Existing Document” on page 5-3.
New	Opens a new document window using the default attributes. See “Creating a New Document” on page 5-2.
New Layout	Displays a panel that lets you change the default attributes used in creating a new document. See “Creating a New Document” on page 5-2.
Save, Save As	Saves a document (consisting of one or more TIFF images) to a TIFF file. See “Saving a Document” on page 5-3.
Revert to Saved	Undoes the changes that have been made since the last time the document was saved.

Commands in the Format Menu

Icon Builder’s Format menu, described in Table 5-3, contains submenus of standard font and text commands, which you can use to affect the appearance of text used in Icon Builder, as well as the Document Layout command.

Table 5-3 Icon Builder’s Format Menu

Command	Description
Font	Opens a menu of standard Font commands, which you use to set the font characteristics of the selected text in the TextTool inspector.
Text	Opens a menu of standard Text commands, which you use to set the attributes of the selected text in the TextTool inspector.
Document Layout	Displays the Document Layout panel, which you use to change the attributes of a document window. See “Changing the Attributes of a Document” on page 5-17.

Commands in the Tools Menu

The Tools menu, described in Table 5-4, contains commands for accessing the primary tools provided in Icon Builder.

Table 5-4 Icon Builder's Tools Menu

Command	Description
Inspector	Opens the Inspector panel, which you use to change the appearance and behavior of the available tools. See "Using the Tools Inspector" on page 5-8.
Tools	Opens the Tools panel, which you use to select among available tools. See "Using Icon Builder Tools" on page 5-4.
Colors	Opens the standard Colors panel, which you use to choose color or gray-scale values.
Obese Bits	Opens the ObeseBits panel, which you use to magnify the contents of a document window. See "Zooming In on a Document" on page 5-15.
Load Tool	Opens the Load Tool panel, which you use to load a user-defined tool.

Navigating the OpenStep API with Header Viewer

6 



Header Viewer is a programmer's research and reference tool. With Header Viewer, you navigate through OpenStep's application programming interface (API); review the declarations of language elements such as methods, functions, and constants; and retrieve relevant passages from the OpenStep developer documentation. Header Viewer helps you unravel unfamiliar code, whether building blocks from the OpenStep software kits, programming examples, or programs written by your own development team.

This chapter introduces Header Viewer and shows how it can speed your programming effort. The next section gives you some insight into how Header Viewer works. The remaining sections teach you how to use Header Viewer in your development work.

Header Viewer and Header Files

Header files are indispensable to developers. They declare the programmatic interfaces to individual modules of code and, through `#import` and `#include` statements, link one header file to another. The header file is the final authority for any particular programmatic interface.

Although the final authority, header files are not always the easiest reference to use. In an object-oriented environment such as OpenStep, a large proportion of header files declare the interfaces to classes, in general, one class per header file. However, since a class inherits part of its interface from its superclass, you might have to search several files to learn the capabilities of a single class. Compounding this problem is the sheer number of classes: The Application Kit alone has over 60 classes. A more sensible approach would be to have a tool that could follow these interconnections directly.

In principle, a simple tool that performed text-based searches on the header file might do the job; however, given that the header files for the Application Kit alone contain several thousand lines of declarations, in practice such a tool would be too slow. A better approach is needed.

Precompiled Headers

Header Viewer's approach is to use OpenStep's *precompiled headers*. A precompiled header has been processed by the preprocessor, leaving it in a compact, binary format. During preprocessing, macros are expanded, comments removed, other headers included as indicated by `#include` or `#import` directives, and language tokens parsed from the resulting stream of text. Because it uses precompiled headers, Header Viewer can be called a *language-sensitive browser*.

Precompiled headers are stored in files having a `.p` extension. OpenStep includes precompiled headers (in `/usr/openstep/Developer/include`) for the Application Kit (`Appkit/Appkit.p`) and FoundationKit (`Foundation/Foundation.p`).

Upon launch, Header Viewer loads the precompiled headers by default. If you plan to scan the API's of the other libraries, you can add the headers. In addition, you can display the contents of other header files, including your own, by adding them to Header Viewer's list. See "Adding Header Files" on page 6-12 for more information.

Note – This chapter refers to both `.h` and `.p` files as header files. Header Viewer accommodates both by precompiling `.h` files when needed.

Language Elements

Header Viewer scans a precompiled header for the language elements it contains and categorizes them by type. It then lets you select the category you want to display. For example, you can look at all the methods, the functions, or the constants that are declared. Table 6-1 describes the categories Header Viewer lists.

Table 6-1 Language Elements You Can Look at with Header Viewer

Language Element	Description
class	In the Objective C language, a prototype for a particular kind of object. A class definition declares instance variables and defines methods for all members of the class. In Header Viewer, classes can be listed or browsed in a class hierarchy.
category	In the Objective C language, a set of method definitions that is segregated from the rest of the class definition. Categories can be used to organize a class definition into parts or to add methods to an existing class without creating a subclass.
protocol	In the Objective C language, a list of methods not associated with any particular class. Protocols are often used to promote reuse of a design.
method	A procedure that can be executed by an object.
function	A routine designed to accomplish a particular task. Header Viewer lists all functions, including C library functions, OpenStep NS functions, and UNIX system calls.
global data	Variables that are visible to all programming modules.
typedef	A type definition. In the C language, used to create a new variable type from existing types.

Table 6-1 Language Elements You Can Look at with Header Viewer (Continued)

Language Element	Description
struct	A structure data type, identified in the C language by the keyword <code>struct</code> .
union	A data type that allows different data names and data types to be assigned to the same storage location.
enum	The C language enumerated data type, consisting of a named set of values with an integer assigned to each member of the set.
enum constants	The constants declared within an enumeration set.
constant-like macros	A macro that is defined and treated like a constant.
function-like macros	A macro that is defined and treated like a function.
macros guard	A C language macro that is used to ensure that each header file is included only once. The Objective C <code>#import</code> declaration makes once-hack macros unnecessary.
predefined macros	Symbol names that are defined by the compiler. Many deal with parameters such as byte swapping and determining a machine's CPU type.
header	A file that contains programming declarations. In Header Viewer, header files can be listed or browsed in a hierarchy.

For additional information about the meaning and use of a language element, refer to Chapter 8, “The Objective C Language,” Chapter 9, “The Objective C Extensions,” or *The C Programming Language* by Kernighan and Ritchie.

Header Viewer and OpenStep Documentation

Header Viewer retrieves documentation for any OpenStep class, method, protocol, or category. It accesses files in `/usr/openstep/Developer/Documentation`, which correspond to chapters in *OpenStep Programming Reference*. It does not retrieve documentation for the other types of API elements, such as those declared in the standard C libraries.

Header Viewer retrieves only the defining passage from the developer documentation, not every reference. For example, when you use it to access the documentation for the `display` method of the `NSView` class, it shows you the description of the method found in the `NSView` class specification. It does not search other sources for additional references to the method.

Using Header Viewer

Header Viewer has a single main window, which can display a Browser view or a Finder view. The window displays the Browser view when the application is launched, but you can set a preference for the Finder view to appear as the default view. You can switch from one view to the other using the Utilities menu.

In either the Browser view or Finder view, you choose to view either a header file or OpenStep developer documentation. A pop-up list in the lower right corner of the Header Viewer main window offers a choice of either a Header File or Documentation. Header files are always available; if documentation is not available, the Documentation choice is grayed out.

The Browser View

The Browser view, shown in Figure 6-1 on page 6-6, outlines hierarchies of classes and lists all occurrences of language elements such as methods or functions. The Browser view helps you discover what is available to you, both in OpenStep's libraries and in code you obtain from others.

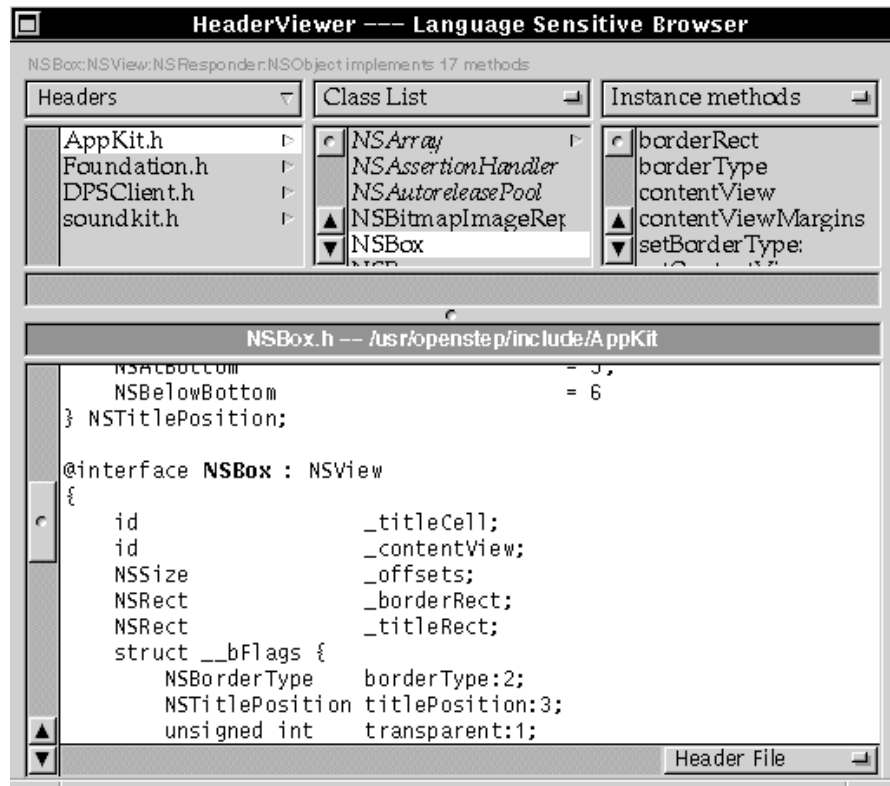


Figure 6-1 Header Viewer's Browser View

Header Viewer's Browser view resembles the Browser view of the OpenStep Workspace Manager's File Viewer. However, with Header Viewer's Browser view, you are not looking at the file system; rather, you are looking at the contents of header files. You do not see ordinary files and folders; instead, you see lists of header files or classes, methods, and other language elements.

A pop-up list above each Browser column allows you to choose what kinds of things you see in the Browser columns. You choose from a list of 14 different language elements (see "Language Elements" on page 6-3).

Note – The pop-up list above a Browser column changes depending on what you select in the preceding column.

The left-most column always shows a list of header files. The pop-up list above the column (see Figure 6-2) lets you remove header files (see “Adding Header Files” on page 6-12 for more information).

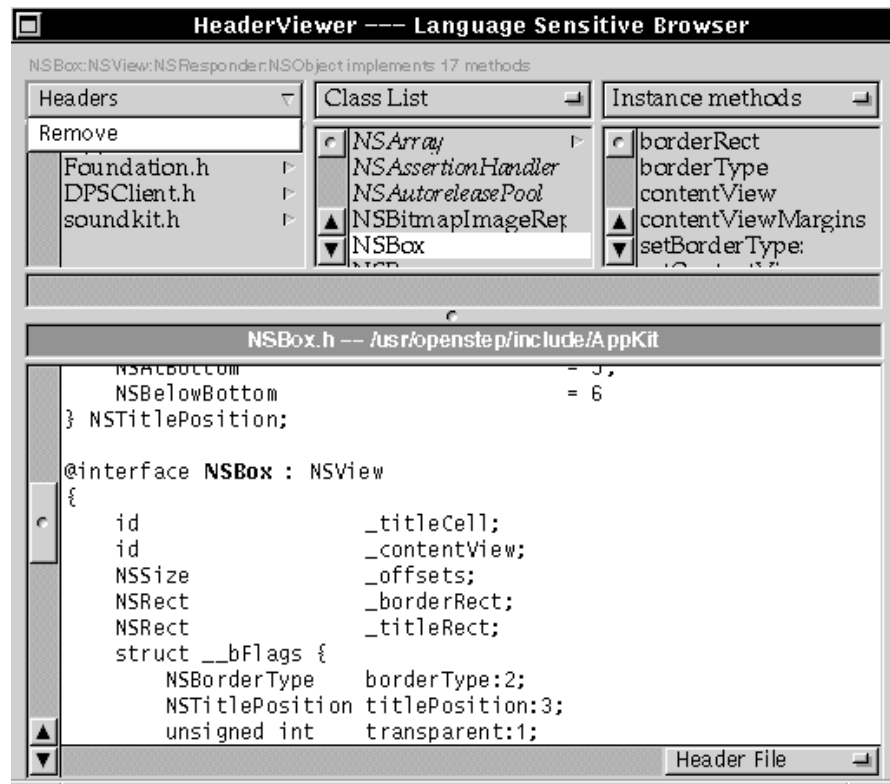


Figure 6-2 Removing a Header File

If you are browsing a Header Hierarchy and select a header that includes other headers (indicated by a pointer in the Browser column), you see a list of included headers in the neighboring column. A pop-up list above the neighboring column allows a choice of Direct Headers or All Headers (see Figure 6-3 on page 6-8). Direct headers are only those you see included in the header file. All headers are all those upon which the header is dependent, whether directly or indirectly.

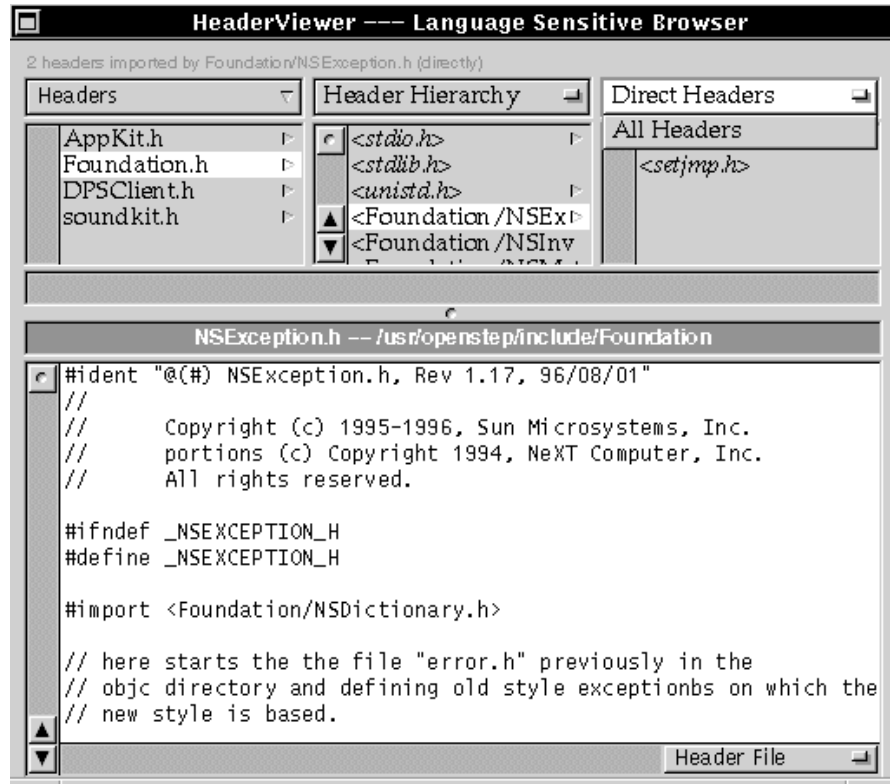


Figure 6-3 Selecting Direct Headers or All Headers in a Header Hierarchy

If you choose Class Hierarchy and select a class listed in the column, you see a list of the subclasses that inherit from the class. Use a pop-up list above the column (see Figure 6-4 on page 6-9) to view superclasses (the parent class from which a class is derived), instance methods, class methods, or categories.

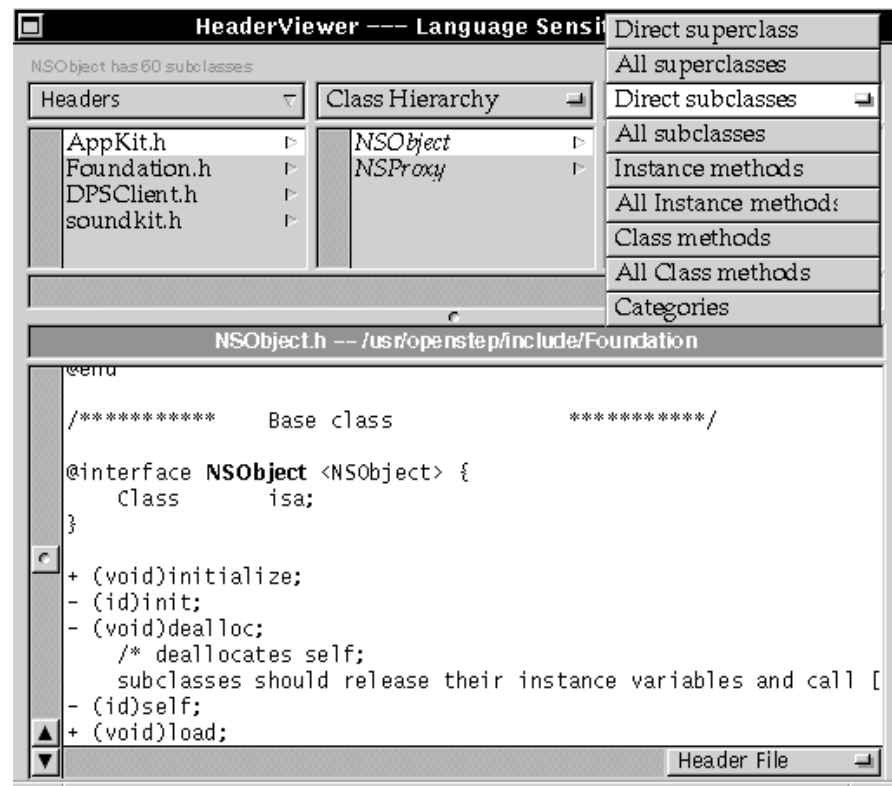


Figure 6-4 Choosing Display in a Class Hierarchy

If you choose Class List and select a class listed in the column, you see a list of its instance methods in the following column. Use the pop-up list above the column to view subclasses, superclasses, class methods, or categories.

If you choose Methods and select a method listed in the column, you see a list of classes in the following column if the method is implemented by more than one class, category, or protocol. Use a pop-up list above the column to view classes, categories, or protocols. If the method is used by only one class, category, or protocol, nothing is shown in the neighboring column.

The Browser view is a powerful aid to visualizing the relationship of one header file to others, the chains of inheritance among classes, and the use of language elements in header files. Its dynamic lists present the information in compact form.

The Finder View

The Finder view lets you search header files and retrieve related documentation. In practice, you use the Finder view, shown in Figure 6-5, to look up the details you need to put any language element to good use.

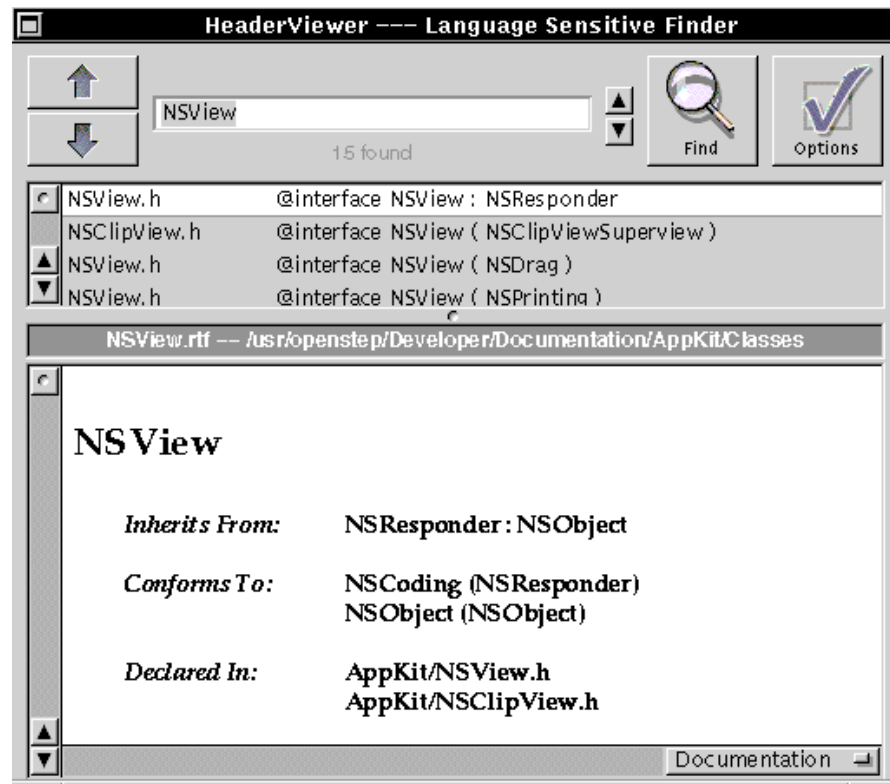


Figure 6-5 Header Viewer's Finder View

The Finder view displays a text field at the top. Entering a string and clicking the Find button initiates a search for the string. Results of a search are displayed in a scrollable list see Figure 6-6 on page 6-11).

Clicking on any item in the results list displays contents of a header file or relevant documentation. The full context of the search string appears in bold when the document window displays a header file. When matching documentation is available, the document display shows the relevant passage from *OpenStep Programming Reference*.

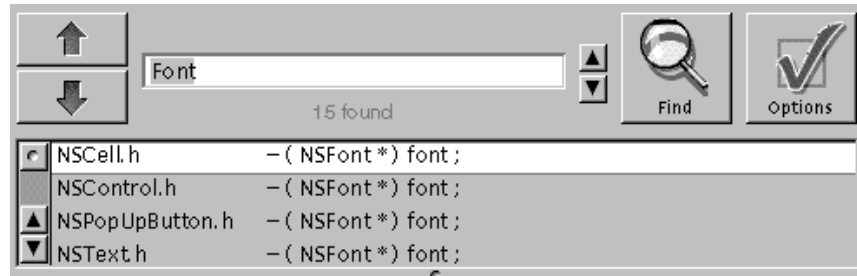


Figure 6-6 Find Results List

You can move through the results list by clicking either of two large arrows to the left of the text field (or by using the up and down arrows on the keyboard). Two small arrows to the right of the text field let you recall search strings that you have previously used. Double-clicking on any item in the results list opens the selected header file in Edit. The search string is highlighted in the Edit window.

An Options button opens the Find Control Options panel, shown in Figure 6-7 on page 6-12) that lets you narrow searches to specific language elements and selected header files. Three buttons under the title Element Usage allow searches to include Declarations, References, or All Tokens. Ordinarily, Header Viewer confines the results of a search to actual declarations of the string. You can broaden the search to references to the string or any appearance of the string in a header file (except in a comment).

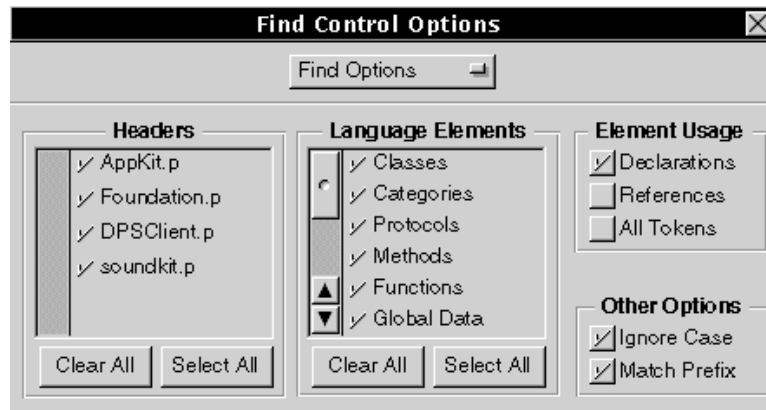


Figure 6-7 Selecting Find Control Options

You can also narrow or broaden searches by choosing Ignore Case or Match Prefix in the Find Control Options panel.

You can alter the rank of the results by choosing Sort Order from the pop-up list in the Find Control Options panel. By manipulating the Sort Order and the Find options, you precisely tune a search to yield only the information most useful to you.

Adding Header Files

On launch, Header Viewer looks for headers that correspond to headers present in the Header Viewer list. By default, Header Viewer uses the headers that correspond to the header files `Appkit.h` and `Foundation.h`.

You add your own header files to Header Viewer by choosing Add Header in the Utilities menu and selecting a file with the standard Open panel. Header Viewer temporarily compiles the header, parses its language elements, and displays its contents in the document window.

Header Viewer retains the precompiled header in memory as long as Header Viewer is running. If you want to continue to use your own headers the next time you launch Header Viewer, you must first recompile the header. Header Viewer's Preferences panel (see Figure 6-9 on page 6-14) allows you to specify which header files will be listed at launch.

The Find Panel

Use the Find in Viewer panel, shown in Figure 6-8, to search for any text string in the Header Viewer document window or Finder view results list.

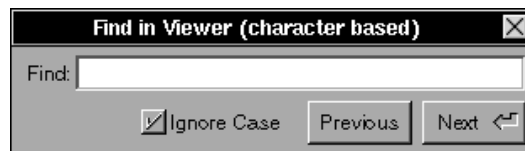


Figure 6-8 Header Viewer's Find in Viewer Panel

This Find panel is similar to the one you use in Edit. Unlike Header Viewer's more powerful Find button, which searches precompiled headers for specific language elements, the Find in Viewer panel is a character-based search tool that only scans the contents of the document window or the Finder view results list. However, this can be very useful when searching for a specific method in a long list of methods displayed in a Finder view results list, or in looking for all occurrences of a string in a header file or documentation chapter.

Header Viewer and the File Viewer

Occasionally you may encounter a header file you would like to investigate while working in the File Viewer. Double-clicking on a `.h` file usually opens it in Edit. To force Header Viewer to open the `.h` file, choose Add Header from the Header Viewer submenu of Workspace Manager's Services menu. Header Viewer temporarily precompiles the header, parses its language elements, and displays its contents in the document window.

Header Viewer and Edit

If you are writing code or debugging source, you will want to access Header Viewer from your editor. Select any text string and pass it to Header Viewer. To do so, choose Find from the Header Viewer submenu of the Services menu.

Note – Header Viewer can match an entire message expression. This is one of Header Viewer’s most powerful facilities. If you are working in Edit, double-click on the first square bracket in an expression to select the entire message expression, then pass it to Header Viewer through the Services menu. Header Viewer strips the expression of its receiver and arguments and matches the message to all corresponding methods. A look at the developer documentation should quickly reveal the correct use of the expression.

Setting Preferences

The Preferences command in the Info menu displays the Preferences panel, shown in Figure 6-9. Preferences options are grouped in three views. A pop-up list offers a choice of preference view: Header Files, Documentation Directories, and Other Options.

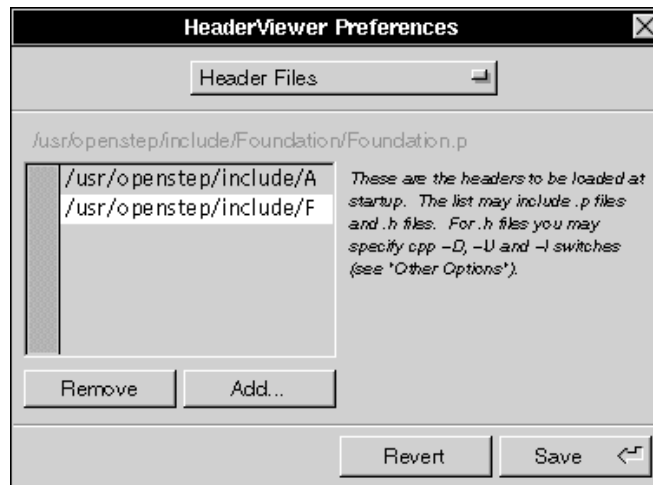


Figure 6-9 Header Viewer Preferences Panel

Header Files Preferences

Choose Header Files from the pop-up list in the Preferences panel to specify additional header files that Header Viewer lists when launched. You can add conventional headers (.h files).

Documentation Preferences

Choose Documentation Directories from the pop-up list in the Preferences panel to enable Header Viewer to access additional documentation directories. Documentation must conform to the format used by OpenStep developer documentation, as indicated in the panel shown in Figure 6-10.

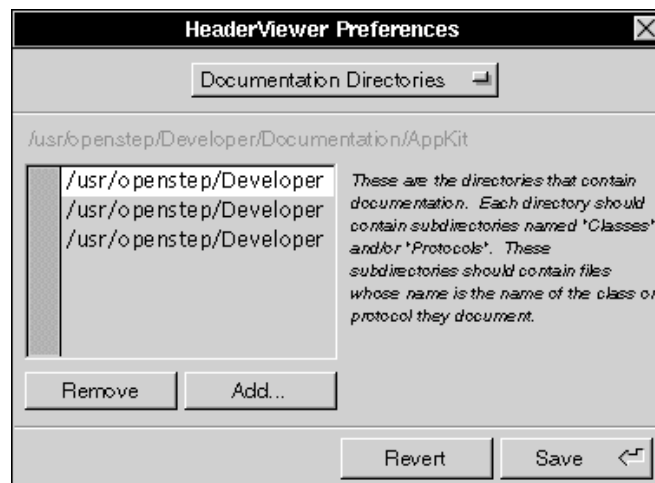


Figure 6-10 Documentation Directories Panel

Other Options Preferences

Choose Other Options from the pop-up list in the Preferences panel to set defaults for Header Viewer on launch and to set default options for the C preprocessor when you add a header file to Header Viewer (see Figure 6-11 on page 6-16).

The Default View group lets you specify what you see when you launch Header Viewer: header files or documentation, the Browser view or the Finder view.

The Default C Preprocessor Options group lets you specify default options (such as the -D, -U, and -I switches) for the C preprocessor to use when precompiling a header file for use in Header Viewer. These options precede any other options you choose when you add a header file to Header Viewer.

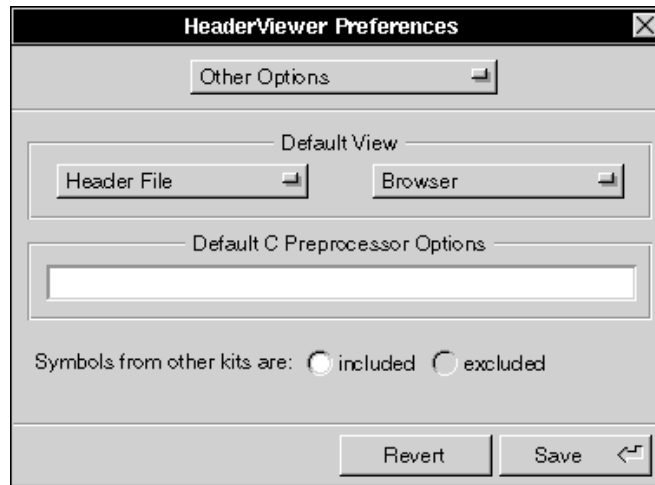


Figure 6-11 Other Options Panel

Header Viewer Command Reference

Header Viewer’s main menu offers the standard Info, Edit, Windows, Services, Print, Hide, and Quit commands. All commands unique to Header Viewer are located in the Find (under Edit) and Utilities menus. These menus and the commands they contain are described in “Commands in the Find Menu” and in “Commands in the Utilities Menu” on page 6-18.

Commands in the Find Menu

The Find menu offers the commands listed in Table 6-2 on page 6-17 for finding text in the document window or in the list of results displayed in the Finder view.

Table 6-2 Find Menu Commands

Command	Description
Find Panel	Opens a panel to allow you to enter a text string for a search.
Find Next	Searches forward from the insertion point or current selection in the list of results displayed in the Finder view. It searches for the text you've typed in the Find field of the Find panel. It does the same thing as the Next button in the Find panel.
Find Previous	Same action as the Find Next command, but searches backward.
Enter Selection	A shortcut; same as copying a text selection and pasting it into the Find panel.

Commands in the Utilities Menu

The Utilities menu contains the commands shown in Table 6-3 that let you control the Header Viewer main window.

Table 6-3 Utilities Menu Commands

Command	Description
Find	Displays the Finder view.
Browse	Displays the Browser view.
Browse Selection	Displays the selection you have made in the Finder view within a Browser view hierarchy.
Update	Determines if any header files have changed since Header Viewer was launched; recompiles and reparses headers that have changed (the precompiled version is stored only in memory). Most useful when you have made changes to your own header files.
Add Header	Allows you to add your own header files to Header Viewer. You can add conventional headers (.h files). If you would like Header Viewer to use these files whenever it is launched, choose the Preferences command from Header Viewer's main menu.
Open in Edit	Opens the current header or documentation files in the Edit application and highlights the declaration of the language element you have specified in the Finder view or selected in the Browser view.

The `NSObject` Class



Characteristic	Description
Inherits From:	none (<code>NSObject</code> is the root class.)
Conforms To	<code>NSObject</code>
Declared In:	<code>Foundation/NSObject.h</code> <code>Foundation/NSRunLoop.h</code>

Class Description

`NSObject` is the root class of all ordinary Objective C inheritance hierarchies; it has no superclass. Its interface derives from two sources: the methods it declares directly and those declared in the `NSObject` protocol. Its interface is divided in this way so that objects inheriting from other root classes (notably `NSProxy`) can stand in for ordinary objects without having to inherit from `NSObject`. The following discussion makes no distinction between the methods declared in this class and those declared in the `NSObject` protocol.

From `NSObject`, other classes inherit a basic interface to the runtime system for the Objective C language. It is through `NSObject` that instances of all classes obtain their ability to behave as objects. Among other things, the `NSObject` class provides inheriting classes with a framework for creating, initializing, deallocating, comparing, and archiving objects, for performing methods selected at run-time, for querying an object about its methods and its position in the inheritance hierarchy, and for forwarding messages to other

objects. For example, to ask an object what class it belongs to, you would send it a `class` message. To find out whether it implements a particular method, you would send it a `respondsToSelector:` message.

The `NSObject` class is an abstract class; programs use instances of classes that inherit from `NSObject`, but never of `NSObject` itself.

Initializing an Object to Its Class

Every object is connected to the runtime system through its `isa` instance variable, inherited from the `NSObject` class, which identifies the object's class; it points to a structure that is compiled from the class definition. Through `isa`, an object can find whatever information it needs at run time, such as its place in the inheritance hierarchy, the size and structure of its instance variables, and the location of the method implementations it can perform in response to messages.

Because all ordinary objects inherit directly or indirectly from the `NSObject` class, they all have this variable. The defining characteristic of an “object” is that its first instance variable is an `isa` pointer to a class structure.

The installation of the class structure—the initialization of `isa`—is one of the responsibilities of the `alloc` and `allocWithZone:` methods, the same methods that create (allocate memory for) new instances of a class. In other words, class initialization is part of the process of creating an object; it is not left to the methods, such as `init`, that initialize individual objects with their particular characteristics.

Instance and Class Methods

Every object requires an interface to the runtime system, whether it is an instance object or a class object. For example, it should be possible to ask either an instance or a class whether it can respond to a particular message. So that this will not mean implementing every `NSObject` method twice, once as an instance method and again as a class method, the run-time system treats methods defined in the root class in a special way: *Instance methods defined in the root class can be performed both by instances and by class objects.*

A class object has access to class methods—those defined in the class and those inherited from the classes above it in the inheritance hierarchy—but generally not to instance methods. However, the runtime system gives all class objects

access to the instance methods defined in the root class. Any class object can perform any root instance method, provided it does not have a class method with the same name.

For example, a class object could be sent messages to perform `NSObject`'s `respondsToSelector:` and `perform:withObject:` instance methods:

```
SEL method = @selector(riskAll:);

if ( [MyClass respondsToSelector:method] )
    [MyClass perform:method withObject:self];
```

When a class object receives a message, the run-time system looks first at the receiver's set of class methods. If it fails to find a class method that can respond to the message, it looks at the set of instance methods defined in the root class. If the root class has an instance method that can respond (as `NSObject` does for `respondsToSelector:` and `perform:withObject:`), the run-time system uses that implementation and the message succeeds.

Only instance methods available to a class object are those defined in the root class. If `MyClass` in the example above had reimplemented either `respondsToSelector:` or `perform:withObject:`, those new versions of the methods would be available only to instances. The class object for `MyClass` could perform only the versions defined in the `NSObject` class. (Of course, if `MyClass` had implemented `respondsToSelector:` or `perform:withObject:` as class methods rather than instance methods, the class would perform those new versions.)

Initializing the Class

Method	Description
+ (void)initialize	Initializes the class before it is used (before it receives its first message).

Creating and Destroying Instances

Method	Description
+ (id)alloc	Returns a new, uninitialized instance of the receiving class
+ (id)allocWithZone:(NSZone *)zone	Returns a new, uninitialized instance of the receiving class in zone.
+ (id)new	Allocates a new instance of the receiving class, sends it an init message, and returns the initialized object returned by init. This method is simply a convenient cover for the alloc and init methods.
- (id)copy	Invokes copyWithZone:. This method is implemented in NSObject as a convenience to subclasses. A subclass need override only copyWithZone: for both copy and copyWithZone: to operate correctly.
- (void)dealloc	Deallocates the memory occupied by the receiver.
- (id)init	Implemented by subclasses to initialize a new object (the receiver) immediately after memory for it has been allocated.
- (id)mutableCopy	Invokes mutableCopyWithZone:. This method is implemented in NSObject as a convenience to subclasses. A subclass need override only mutableCopyWithZone: for both mutableCopy: and mutableCopyWithZone: to operate correctly.

Identifying Classes

Method	Description
+ (Class)class	Returns <code>self</code> . Since this is a class method, it returns the class object.
+ (Class)superclass	Returns the class object for the receiver's superclass.

Testing Class Functionality

Method	Description
+ (BOOL)instancesRespondToSelector:(SEL)aSelector	Returns YES if instances of the class are capable of responding to <code>aSelector</code> messages, and NO if they are not.

Testing Protocol Conformance

Method	Description
+ (BOOL)conformsToProtocol:(Protocol *)aProtocol	Returns YES if the receiving class conforms to aProtocol, and NO if it does not.

Obtaining Method Information

Method	Description
+ (IMP)instanceMethodForSelector:(SEL)aSelector	Locates and returns the address of the implementation of the aSelector instance method.
- (IMP)methodForSelector:(SEL)aSelector	Locates and returns the address of the receiver's implementation of the aSelector method, so that it can be called as a function.
- (NSMethodSignature *) methodSignatureForSelector:(SEL)aSelector	Returns an object that contains a description of the aSelector method, or nil if the aSelector method can not be found.

Describing Objects

Method	Description
+ (NSString *)description	Subclasses override this method to return a human-readable string representation of the contents of the receiver. <code>NSObject</code> 's implementation simply prints the name of the receiver's class.

Posing

Method	Description
+ (void)poseAsClass:(Class)aClassObject	Causes the receiving class to “pose as” its superclass.

Error Handling

Method	Description
- (void)doesNotRecognizeSelector:(SEL)aSelector	Handles <code>aSelector</code> messages that the receiver does not recognize.

Sending Deferred Messages

Method	Description
+(void)cancelPreviousPerformRequestsWithTarget:(id)aTarget selector:(SEL)aSelector object:(id)anObject	Cancels previous perform requests having the same target and argument (as determined by isEqual:), and the same selector. This method removes timers only in the current run loop, not all run loops.
-(void)performSelector:(SEL)aSelector object:(id)anObject afterDelay:(NSTimeInterval)delay	Sends an aSelector message to anObject after delay. self and anObject are retained until after the action is executed.

Forwarding Messages

Method	Description
-(void)forwardInvocation:(NSInvocation *)anInvocation	Implemented by subclasses to forward messages to other objects.

Archiving

Method	Description
- (id)awakeAfterUsingCoder: (NSCoder *)aDecoder	Implemented by subclasses to reinitialize the receiver after unarchiving. The <code>NSObject</code> implementation of this method simply returns <code>self</code> .
- (Class)classForArchiver	Identifies the class to be used during archiving. <code>NSObject</code> 's implementation returns the object returned by <code>classForCoder:</code> .
- (Class)classForCoder	Identifies the class to be used during serialization. An <code>NSObject</code> returns its own class by default.
- (id)replacementObjectForArchiver: (NSArchiver *)anArchiver	Allows an object to substitute another object for itself during archiving. <code>NSObject</code> 's implementation returns the object returned by <code>replacementObjectForCoder:</code> .
- (id)replacementObjectForCoder: (NSCoder *)anEncoder	Allows an object to substitute another object for itself during serialization. <code>NSObject</code> 's implementation returns <code>self</code> .
+ (void)setVersion:(int)version	Sets the class version number to <code>version</code> .
+ (int)version	Returns the version of the class definition.

The Objective C Language



This chapter describes the OpenStep Objective C language as well as the principles of object-oriented programming as implemented in Objective C. Chapter 9, “The Objective C Extensions” continues the discussion by taking up more advanced and less commonly used language features.

Objective C syntax is a superset of standard C/C++ syntax, and its compiler works for C, C++, and Objective C source code. The compiler recognizes Objective C source files by a `.m` extension, just as it recognizes files containing only standard C syntax by a `.c` extension or C++ with a `.cc` extension. As implemented for OpenStep, the Objective C language is fully compatible with ANSI standard C.

Objective C can also be used as an extension to C++. This may seem superfluous since C++ is itself an object-oriented extension of C. But C++ was designed primarily as “a better C,” and not necessarily as a full-featured object-oriented language. It lacks some of the possibilities for object-oriented design that dynamic typing and dynamic binding bring to Objective C. At the same time, it has useful language features not found in Objective C. When you use the two languages in combination, you can assign appropriate roles to the features found in each and take advantage of what is best in both. The *NEO Programming Guide* has more on combining C++ with Objective C.

Because object-oriented programs postpone many decisions from compile time to run time, object-oriented languages depend on a run-time system for executing the compiled code. This chapter and Chapter 9, “The Objective C Extensions” present the language, but touch on important elements of the run-

time system as they are important for understanding language features. The CAFE compiler has been modified to also compile Objective C and provide its own run-time system.

Objects

As the name implies, object-oriented programs are built around *objects*. An object associates data with the particular operations that can use or affect that data. In Objective C, these operations are known as the object's *methods*; the data they affect are its *instance variables*. In essence, an object bundles a data structure (instance variables) and a group of procedures (methods) into a self-contained programming unit.

For example, through the OpenStep Application Kit, you can produce an object that displays a matrix of cells to users of your application. The cells might be text fields where the user can enter data, a series of mutually exclusive switches, a list of buttons or menu commands, or a bank of sliders. Figure 8-1 illustrates some of the different kinds of cells a matrix can contain:

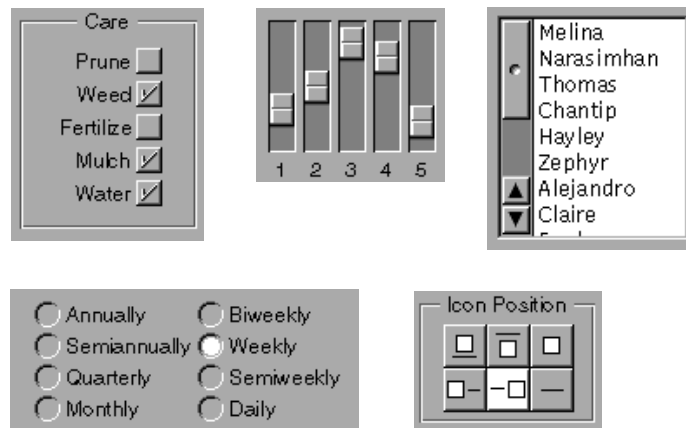


Figure 8-1 Some NSMatrix Objects

An `NSMatrix` object has instance variables that define the matrix, including its dimensions and coordinates, the font used to display character strings in the cells, the arrangement of cells into rows and columns, and what to do when a cell

is selected. An `NSMatrix` also has methods that do such things as alter its size, change its position on-screen, add and remove cells, highlight a particular cell, and set the color that is displayed between cells.

Each cell in an `NSMatrix` is also an object. Cells have instance variables that record their contents and what action to take when the cell is selected. They have methods to determine what the cell looks like and to track the cursor as it moves from cell to cell.

In Objective C, an object's instance variables are internal to the object; you get access to an object's state through the object's methods. For others to find out something about an object, there has to be a method to supply the information. For example, an `NSMatrix` has methods that reveal its size, the currently selected cell, and the current number of columns and rows.

Moreover, an object sees only the methods that were designed for it; it cannot mistakenly perform methods intended for other types of objects. Just as a C function protects its local variables, hiding them from the rest of the program, an object hides both its instance variables and its method implementations.

The `id` Data Type

In Objective C, objects are identified by a distinct data type, `id`. This type is defined as a pointer to an object—in reality, a pointer to the object's data (its instance variables). Like a C function or an array, an object is identified by its address. All objects, regardless of their instance variables or methods, are of type `id`.

```
id anNSObject;
```

For the object-oriented constructs of Objective C, such as method return values, `id` replaces `int` as the default data type. For strictly C constructs, such as function return values, `int` remains the default type.

The keyword `nil` is defined as a null object, an `id` with a value of 0. The data types `id` and `nil`, and the other basic types of Objective C are defined in the header file `objc.h`, which is located in the `objc` subdirectory of `/usr/openstep/include`.

Dynamic Typing

The `id` type is completely nonrestrictive. By itself, it yields no information about an object, except that it is an object.

But objects are not all the same. An `NSMatrix` will not have the same methods or instance variables as an object that represents one of its cells. Cells that display buttons (`NSButtonCells`) will not be exactly like those that display text (`NSTextFieldCells`). At some point, a program needs to find more specific information about the objects it contains—what the object’s instance variables are, what methods it can perform, and so on. Since the `id` type designator cannot supply this information to the compiler, each object has to be able to supply it at run time.

This is possible because every object carries with it an `isa` instance variable that identifies the object’s *class*—what kind of object it is. Every `NSMatrix` object would be able to tell the run-time system that it is an `NSMatrix`. Every `NSButtonCell` can say that it is an `NSButtonCell`. Objects with the same behavior (methods) and the same kinds of data (instance variables) are members of the same class.

Objects are thus *dynamically typed* at run time. Whenever it needs to, the run-time system can find the exact class that an object belongs to, just by asking the object. Dynamic typing in Objective C serves as the foundation for dynamic binding, discussed in “Dynamic Binding” on page 8-7.

The `isa` pointer also enables objects to introspect about themselves as objects. The compiler does not discard much of the information it finds in source code; it arranges most of it in data structures for the run-time system to use. Through `isa`, objects can find this information and reveal it at run time. An object can, for example, say whether it has a particular method in its repertoire and what the name of its superclass is.

Object classes are discussed in more detail under “Classes” on page 8-8.

Note – It is also possible to give the compiler information about the class of an object by statically typing it in source code using the class name. Classes are particular kinds of objects, and the class name can serve as a type name. See “Class Types” on page 8-13 and “Static Options” on page 9-22.

Messages

To get an object to do something, you send it a *message* telling it to apply a method. In Objective C, *message expressions* are enclosed in square brackets:

```
[receiver message]
```

The receiver is an object, and the message tells it what to do. In source code, the message is simply the name of a method and any arguments that are passed to it. When a message is sent, the run-time system selects the appropriate method from the receiver's repertoire and invokes it.

For example, the following message tells the `myNSMatrix` object to perform the `display` method, inherited from `NSView`, which draws the matrix and its cells in a window:

```
[myNSMatrix display];
```

Methods can also take arguments. For example, the following message might tell `myNSMatrix` to change its location within the window to coordinates (30.0, 50.0):

```
[myNSMatrix moveTo:30.0 :50.0];
```

Here the method name, `moveTo::`, has two colons, one for each of its arguments. The arguments are inserted after the colons, breaking the name apart. Colons do not have to be grouped at the end of a method name, as they are here. Usually a keyword describing the argument precedes each colon. The `getRow:column:ofCell:` method, for example, takes three arguments:

```
int row, column;
[myNSMatrix getRow:&row column:&column ofCell:someCell];
```

This method finds `someCell` in the matrix and puts the row and column where it is located in the two variables `&row` and `&column`.

Methods that take a variable number of arguments are also possible, though they are somewhat rare. Extra arguments are separated by commas after the end of the method name. (Unlike colons, the commas are not considered part of the name.) In the following example, the imaginary `makeGroup:` method is passed one required argument (`group`) and three that are optional:

```
[receiver makeGroup:group, memberOne, memberTwo, memberThree];
```

Like standard C functions, methods can return values. The following example assigns the identifying integer returned by the `tag` method inherited from `NSControl`, to a variable also named `tag`.

```
int tag;  
tag = [myNSMatrix tag];
```

Note that a variable and a method can have the same name.

One message can be nested inside another. Here the `selectedCell` method returns an object that then receives a `tag` message:

```
int tag = [[myNSMatrix selectedCell] tag];
```

A message to `nil` also is valid,

```
[nil display];
```

but it has no effect. The code is harder to read because it is not apparent at the point of invocation that the receiver can be 'nil'. Also because the method is not invoked at all, it cannot check for the 'nil' case and take appropriate actions. This may make it more difficult to find bugs. Messages to `nil` simply return `nil`.

Polymorphism

As the examples in “Messages” on page 8-5 illustrate, messages in Objective C appear in the same syntactic positions as function calls in standard C. However, because methods “belong to” an object, messages behave differently than function calls.

In particular, an object has access only to the methods that were defined for it. It cannot confuse them with methods defined for other kinds of objects, even if another object has a method with the same name. This means that two objects can respond differently to the same message. For example, each kind of object sent a `display` message could display itself in a unique way. An `NSButtonCell` and an `NSTextFieldCell` would respond differently to identical instructions to track the cursor.

This feature, referred to as *polymorphism*, plays a significant role in the design of object-oriented programs. Together with dynamic binding, it permits you to write code that might apply to any number of different kinds of objects, without your having to choose at the time you write the code what kinds of objects they might be. They might even be objects that will be developed later, by other programmers working on other projects. If you write code that sends a `display` message to an `id` variable, any object that has a `display` method is a potential receiver.

Dynamic Binding

A crucial difference between function calls and messages is that a function and its arguments are joined together in the compiled code, but a message and a receiving object are not united until the program is running and the message is sent. Therefore, the exact method that will be invoked to respond to a message can only be determined at run time, not when the code is compiled.

The precise method that a message invokes depends on the receiver. Different receivers may have different method implementations for the same method name (polymorphism). For the compiler to find the right method implementation for a message, it would have to know what kind of object the receiver is—what class it belongs to. This is information the receiver is able to reveal at run time when it receives a message (dynamic typing), but it is not available from the type declarations found in source code.

The selection of a method implementation happens at run time. When a message is sent, a run-time messaging routine looks at the receiver and at the method named in the message. It locates the receiver's implementation of a method matching the name, “calls” the method, and passes it a pointer to the receiver's instance variables. (For more on this routine, see “How Messaging Works” on page 8-31.)

The method name in a message thus serves to “select” a method implementation. For this reason, method names in messages are often referred to as *selectors*.

This *dynamic binding* of methods to messages works hand in hand with polymorphism to give object-oriented programming much of its flexibility and power. Since each object can have its own version of a method, a program can achieve a variety of results, not by varying the message itself, but by varying just the object that receives the message. This can be done as the program runs; receivers can be decided “on the fly” and can be made dependent on external factors such as user actions.

In the Application Kit, for example, users determine which objects receive messages from menu commands like Cut, Copy, and Paste. The message goes to whatever object controls the current selection. An object that displays editable text would react to a `copy:` message differently than an object that displays scanned images. An `NSMatrix` would respond differently than an `NSCell`. Since messages do not select methods (methods are not bound to messages) until run time, these differences are isolated in the methods that respond to the

message. The code that sends the message does not have to be concerned with them; it does not even have to enumerate the possibilities. Each application can invent its own objects that respond in their own way to `copy:` messages.

Objective C takes dynamic binding one step further and allows even the message that is sent (the method selector) to be a variable that is determined at run time. This is discussed in “How Messaging Works” on page 8-31.

Classes

An object-oriented program is typically built from a variety of objects. A program based on the OpenStep software kits might use `NSMatrix` objects, `NSWindow` objects, `NSArray` objects, `NSText` objects, and many others. Programs often use more than one object of the same kind or *class*—several `NSArray`s or `NSWindows`, for example.

In Objective C, you define objects by defining their classes. The class definition is a prototype for a kind of object; it declares the instance variables that become part of every member of the class, and it defines a set of methods that all objects in the class can use.

The compiler creates just one accessible object for each class, a *class object* that knows how to build new objects belonging to the class. (For this reason it is sometimes also called a “factory object.”) The class object is the compiled version of the class; the objects it builds are *instances* of the class. The objects that will do the main work of your program are instances created by the class object at run time.

All instances of a class have access to the same set of methods, and they all have a set of instance variables cut from the same mold. Each object gets its own instance variables, but the methods are shared.

By convention, class names begin with an uppercase letter (such as `NSMatrix`); the names of instances typically begin with a lowercase letter (such as `myNSMatrix`).

Inheritance

Class definitions are additive; each new class that you define is based on another class through which it *inherits* methods and instance variables. The new class simply adds to or modifies what it inherits. It does not need to duplicate inherited code.

Inheritance links all classes together in a hierarchical tree with a single class, the `NSObject` class, at its root. Every class (but `NSObject`) has a *superclass* one step nearer the root, and any class (including `NSObject`) can be the superclass for any number of *subclasses* one step farther from the root. Figure 8-2 illustrates the hierarchy for a few of the classes in the OpenStep Application Kit.

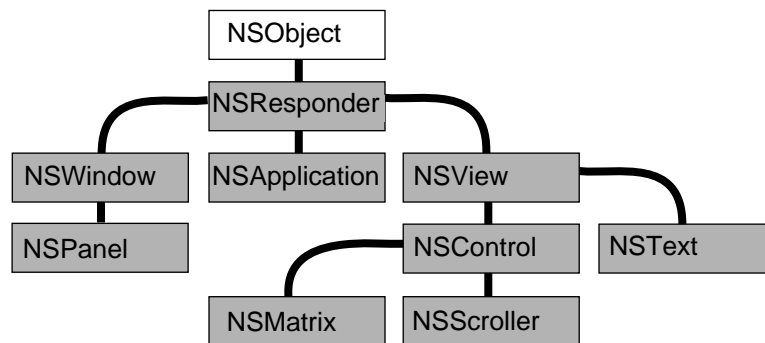


Figure 8-2 Some Application Kit Classes

This figure shows that the `NSMatrix` class is a subclass of the `NSControl` class, the `NSControl` class is a subclass of `NSView`, `NSView` is a subclass of `NSResponder`, and `NSResponder` is a subclass of `NSObject`. Inheritance is cumulative. So an `NSMatrix` object has the methods and instance variables defined for `NSControl`, `NSView`, `NSResponder`, and `NSObject`, as well as those defined specifically for `NSMatrix`. This is simply to say that an `NSMatrix` object is not only an `NSMatrix`, it is also an `NSControl`, an `NSView`, an `NSResponder`, and an `NSObject`.

Every class (but `NSObject`) can thus be seen as a specialization or an adaptation of another class. Each successive subclass further modifies the cumulative total of what is inherited. The `NSMatrix` class defines only the minimum needed to turn an `NSControl` into an `NSMatrix`.

When you define a class, you link it to the hierarchy by declaring its superclass; every class you create must be the subclass of another class (unless you define a new root class). Plenty of potential superclasses are available. The OpenStep development environment includes the `NSObject` class and several software kits containing definitions for more than 125 different classes. Some are classes that you can use “off the shelf”—incorporate into your program as is. Others you might want to adapt to your own needs by defining a subclass.

Some kit classes define almost everything you need, but leave some specifics to be implemented in a subclass. You can thus create very sophisticated objects by writing only a small amount of code, and reusing work done by the programmers at SunSoft.

The NSObject Class

`NSObject` is the only class without a superclass, and the only one that is in the inheritance path for every other class. That is because it defines the basic framework for Objective C objects and object interactions. It imparts to the classes and instances that inherit from it the ability to behave as objects and cooperate with the run-time system.

A class that does not need to inherit any special behavior from another class is nevertheless made a subclass of the `NSObject` class. Instances of the class must at least have the ability to behave like Objective C objects at run time. Inheriting this ability from the `NSObject` class is much simpler and much more reliable than reinventing it in a new class definition.

Chapter 7, “The NSObject Class” has a full specification of the root class and describes its methods in detail.

Note – Implementing a new root class is a delicate task and one with many hidden hazards. The class must duplicate much of what the `NSObject` class does, such as allocate instances, connect them to their class, and identify them to the run-time system. It is strongly recommended that you use the `NSObject` class provided with OpenStep as the root class. This manual does not explain all the ins and outs that you would need to know to replace it.

Inheriting Instance Variables

When a class object creates a new instance, the new object contains not only the instance variables that were defined for its class, but also the instance variables defined for its superclass, and for its superclass's superclass, all the way back to the root `NSObject` class. The `isa` instance variable defined in the `NSObject` class becomes part of every object; it connects each object to its class.

A class does not have to declare instance variables. It can simply define new methods and rely on the instance variables it inherits, if it needs any instance variables at all.

Inheriting Methods

An object has access not only to the methods that were defined for its class, but also to methods defined for its superclass, and for its superclass's superclass, all the way back to the root of the hierarchy. An `NSMatrix` object can use methods defined in the `NSControl`, `NSView`, `NSResponder`, and `NSObject` classes as well as methods defined in its own class.

Any new class you define in your program can therefore make use of the code written for all the classes above it in the hierarchy. This type of inheritance is a major benefit of object-oriented programming. When you use one of the object-oriented kits provided by OpenStep, your programs can take advantage of all the basic functionality coded into the kit classes. You have to add only the code that customizes the kit to your application.

Class objects also inherit from the classes above them in the hierarchy. But because they do not have instance variables (only instances do), they inherit only methods.

Overriding One Method with Another

There is one useful exception to inheritance: When you define a new class, you can implement a new method with the same name as one defined in a class farther up the hierarchy. The new method overrides the original; instances of the new class will perform it rather than the original, and subclasses of the new class will inherit it rather than the original.

For example, the `NSResponder` class defines a `performKeyEquivalent:` method that `NSMatrix` overrides by defining its own version of `performKeyEquivalent:`. The `NSResponder` method is available to all kinds of objects that inherit from the `NSResponder` class—but not to `NSMatrix` objects, which instead perform the `NSMatrix` version of `display`.

Although overriding a method blocks the original version from being inherited, other methods defined in the new class can skip over the redefined method and find the original (see “Messages to self and super” on page 8-39 to learn how).

A redefined method can also incorporate the very method it overrides. When it does, the new method serves only to refine or modify the method it overrides, rather than replace it outright. When several classes in the hierarchy define the same method, but each new version incorporates the version it overrides, the implementation of the method is effectively spread over all the classes.

Although a subclass can override inherited methods, it cannot override inherited instance variables. Since an object has memory allocated for every instance variable it inherits, you cannot override an inherited variable by declaring a new one with the same name. If you try, the compiler complains.

Abstract Classes

Some classes are designed only so that other classes can inherit from them. These *abstract classes* group methods and instance variables that will be used by a number of different subclasses into a common definition. The abstract class is incomplete by itself, but contains useful code that reduces the implementation burden of its subclasses.

The `NSObject` class is the prime example of an abstract class. Although programs often define `NSObject` subclasses and use instances belonging to the subclasses, they never use instances belonging directly to the `NSObject` class. An `NSObject` instance would not be good for anything; it would be a generic object with the ability to do nothing in particular.

In the OpenStep software kits, abstract classes often contain code that helps define the structure of an application. When you create subclasses of these classes, instances of your new classes fit effortlessly into the application structure and work automatically with other kit objects.

Class Types

A class definition is a specification for a kind of object. The class, in effect, defines a data type. The type is based not just on the data structure the class defines (instance variables), but also on the behavior included in the definition (methods).

A class name can appear in source code wherever a type specifier is permitted in C. The following example uses the class name `NSMatrix` as an argument to the `sizeof` operator:

```
int i = sizeof(NSMatrix);
```

Static Typing

You can use a class name in place of `id` to designate an object's type:

```
NSMatrix *myNSMatrix;
```

Since this way of declaring an object type gives the compiler information about what kind of object it is, it is known as *static typing*. Just as `id` is defined as a pointer to an object, objects are statically typed as pointers to a class. Objects are always typed by a pointer. Static typing makes the pointer explicit; `id` hides it.

Static typing permits the compiler to do some type checking—for example, to warn if an object receives a message that it appears not to be able to respond to—and to loosen some restrictions that apply to objects generically typed `id`. In addition, it can make your intentions clearer to others who read your source code. However, it does not defeat dynamic binding or alter the dynamic determination of a receiver's class at run time.

An object can be statically typed to its own class or to any class from which it inherits. For example, since inheritance makes an `NSMatrix` a kind of `NSView`, an `NSMatrix` instance could be statically typed to the `NSView` class, as shown in the following example.

```
NSView *myNSMatrix;
```

This is possible because an `NSMatrix` is an `NSView`. It is more than an `NSView` since it also has the instance variables and method capabilities of an `NSControl` and an `NSMatrix`, but it is an `NSView` nonetheless. For purposes of type checking, the compiler will consider `myNSMatrix` to be an `NSView`, but at run time it will be treated as an `NSMatrix`.

See “Static Options” on page 9-22 for more on static typing and its benefits.

Type Introspection

Instances can reveal their types at run time. The `isMemberOfClass:` method, defined in the `NSObject` class, checks whether the receiver is an instance of a particular class:

```
if ( [anNSObject isMemberOfClass:someClass] )
    . . .
```

The `isKindOfClass:` method, also defined in the `NSObject` class, checks more generally whether the receiver inherits from or is a member of a particular class (whether it has the class in its inheritance path):

```
if ( [anNSObject isKindOfClass:someClass] )
    . . .
```

The set of classes for which `isKindOfClass:` returns YES is the same set to which the receiver can be statically typed.

Introspection is not limited to type information. Later sections of this chapter discuss methods that return the class object, report whether an object can respond to a message, and reveal other information.

See Chapter 7, “The NSObject Class” for more on `isKindOfClass:`, `isMemberOfClass:`, and kindred methods.

Class Objects

A class definition contains various kinds of information, much of it about instances of the class:

- The name of the class and its superclass
- A template describing a set of instance variables
- The declaration of method names and their return and argument types
- The method implementations

This information is compiled and recorded in data structures made available to the run-time system. The compiler creates just one object, a *class object*, to represent the class. The class object has access to all the information about the class, which means mainly information about what instances of the class are like. It is able to produce new instances according to the plan put forward in the class definition.

Although a class object keeps the prototype of a class instance, it is not an instance itself. It has no instance variables of its own and it cannot perform methods intended for instances of the class. However, a class definition can include methods intended specifically for the class object—*class methods* as opposed to *instance methods*. A class object inherits class methods from the classes above it in the hierarchy, just as instances inherit instance methods.

In source code, the class object is represented by the class name. In the following example, the `NSMatrix` class returns the class version number using a method inherited from the `NSObject` class:

```
int versionNumber = [NSMatrix version];
```

However, the class name stands for the class object only as the receiver in a message expression. Elsewhere, you need to ask an instance or the class to return the class id. Both respond to a class message, as shown by the instance of an `NSObject` and the class `NSMatrix` in the following code:

```
id aClass = [anNSObject class];
id matrixClass = [NSMatrix class];
```

As these examples show, class objects can, like all other objects, be typed `id`. But class objects can also be more specifically typed to the `Class` data type, as shown in the following example.

```
Class aClass = [anObject class];
Class matrixClass = [NSMatrix class];
```

All class objects are of type `Class`. Using this type name for a class is equivalent to using the class name to statically type an instance.

Class objects are thus full-fledged objects that can be dynamically typed, receive messages, and inherit methods from other classes. They are special only in that they are created by the compiler, lack data structures (instance variables) of their own other than those built from the class definition, and are the agents for producing instances at run time.

Note – The compiler also builds a “metaclass object” for each class. It describes the class object just as the class object describes instances of the class. But while you can send messages to instances and to the class object, the metaclass object is used only internally by the run-time system.

Creating Instances

A principal function of a class object is to create new instances. The following code tells the `NSMatrix` class to create a new `NSMatrix` instance and assign it to the variable:

```
id myNSMatrix;  
myNSMatrix = [NSMatrix alloc];
```

The `alloc` method dynamically allocates memory for the new object's instance variables and initializes them all to 0—all, that is, except the `isa` variable that connects the new instance to its class. For an object to be useful, it generally needs to be more completely initialized. That is the function of an `init` method. Initialization typically follows immediately after allocation, as in the following example.

```
myNSMatrix = [[NSMatrix alloc] init];
```

This line of code, or one like it, would be necessary before `myNSMatrix` could receive any of the messages that were illustrated in previous examples in this chapter. The `alloc` method returns a new instance and that instance performs an `init` method to set its initial state. Every class object has at least one method (like `alloc`) that enables it to produce new objects, and every instance has at least one method (like `init`) that prepares it for use. Initialization methods often take arguments to allow particular values to be passed and have keywords to label the arguments. For example, the method (`initWithFrame:mode:cellClass:numberOfRows:numberOfColumns:`, would most often initialize a new `NSMatrix` instance), but all initialization methods begin with `init`.

Customization with Class Objects

It is not just a whim of the Objective C language that classes are treated as objects. It is a choice that has intended, and sometimes surprising, benefits for design. It is possible, for example, to customize an object with a class, where the class belongs to an open-ended set. In the Application Kit, an `NSMatrix` object can be customized with a particular kind of `NSCell`.

An `NSMatrix` can take responsibility for creating the individual objects that represent its cells. It can do this when the `NSMatrix` is first initialized and later when new cells are needed. The visible matrix that an `NSMatrix` object draws

on-screen can grow and shrink at run time, perhaps in response to user actions. When it grows, the `NSMatrix` needs to be able to produce new objects to fill the new slots that are added.

But what kind of objects should they be? Each `NSMatrix` displays just one kind of `NSCell`, but there are many different kinds. The inheritance hierarchy in Figure 8-3 shows some of those provided by the Application Kit. All inherit from the generic `NSCell` class.

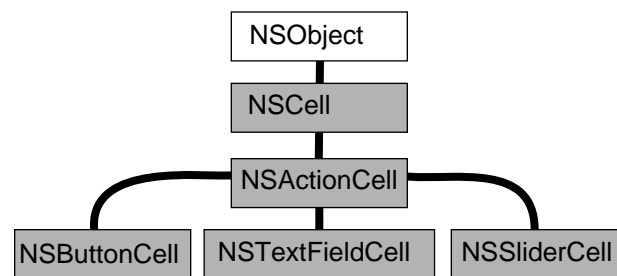


Figure 8-3 Inheritance Hierarchy for Cells

When an `NSMatrix` creates new `NSCell` objects, should they be `NSButtonCells` to display a bank of buttons or switches, `NSTextFieldCells` to display a field where the user can enter and edit text, or some other kind of `NSCell`? The `NSMatrix` must allow for any kind of `NSCell`, even types that have not been invented yet.

One solution to this problem would be to define the `NSMatrix` class as an abstract class and require everyone who uses it to declare a subclass and implement the methods that produce new cells. Because they would be implementing the methods, users of the class could be sure that the objects they created were of the right type.

But this requires others to do work that ought to be done in the `NSMatrix` class, and it unnecessarily proliferates the number of classes. Since an application might need more than one kind of `NSMatrix`, each with a different kind of `NSCell`, it could become cluttered with `NSMatrix` subclasses. Every time you invented a new kind of `NSCell`, you would also have to define a new kind of `NSMatrix`. Moreover, programmers on different projects would be writing virtually identical code to do the same job, all to make up for `NSMatrix`'s failure to do it.

A better solution, the solution the `NSMatrix` class actually adopts, is to allow `NSMatrix` instances to be initialized with a kind of `NSCell`—with a class object. It defines a `setCellClass:` method that passes the class object for the kind of `NSCell` object an `NSMatrix` should use to fill empty slots, as in the following example.

```
[myNSMatrix setCellClass:[NSButtonCell class]];
```

The `NSMatrix` uses the class object to produce new cells when it is first initialized and whenever it is resized to contain more cells. This kind of customization would be impossible if classes were not objects that could be passed in messages and assigned to variables.

Variables and Class Objects

When you define a new class of objects, you can decide what instance variables they should have. Every instance of the class will have its own copy of all the variables you declare; each object controls its own data.

However, you cannot prescribe variables for the class object; there are no “class variable” counterparts to instance variables. Only internal data structures, initialized from the class definition, are provided for the class. The class object also has no access to the instance variables of any instances; it cannot initialize, read, or alter them.

Therefore, for all the instances of a class to share data, an external variable of some sort is required. Some classes declare static variables and provide class methods to manage them. Declaring a variable static in the same file as the class definition limits its scope to just the class—and to just the part of the class that is implemented in the file. Unlike instance variables, static variables cannot be inherited by subclasses, unless the subclasses are defined in the same file.

Static variables help give the class object more functionality than just that of a “factory” producing instances; it can approach being a complete and versatile object in its own right. A class object can be used to coordinate the instances it creates, dispense instances from lists of objects already created, or manage other processes essential to the application. In the limiting case, when you need only one object of a particular class, you can put all the object’s state into static variables and use only class methods. This saves the step of allocating and initializing an instance.

Note – It would also be possible to use external variables that were not declared `static`, but the limited scope of static variables better serves the purpose of encapsulating data into separate objects.

Initializing a Class Object

If a class object is to be used for anything besides allocating instances, it may need to be initialized just as an instance is. Although programs do not allocate class objects, Objective C does provide a way for programs to initialize them.

The run-time system sends an `initialize` message to every class object before the class receives any other messages. This gives the class a chance to set up its run-time environment before it is used. If no initialization is required, you do not need to write an `initialize` method to respond to the message; the `NSObject` class defines an empty version that your class can inherit and perform.

If a class makes use of static or global variables, the `initialize` method is a good place to set their initial values. For example, if a class maintains an array of instances, the `initialize` method could set up the array and even allocate one or two default instances to have them ready.

Methods of the Root Class

All objects, classes and instances alike, need an interface to the run-time system. Both class objects and instances should be able to introspect about their abilities and to report their place in the inheritance hierarchy. It is the province of the `NSObject` class to provide this interface.

So that `NSObject`'s methods will not all have to be implemented twice—once to provide a run-time interface for instances and again to duplicate that interface for class objects—class objects are given special dispensation to perform instance methods defined in the root class. When a class object receives a message that it cannot respond to with a class method, the run-time system will see if there is a root instance method that can respond. The only instance methods that a class object can perform are those defined in the root class, and only if there is no class method that can do the job.

For more on this peculiar ability of class objects to perform root instance methods, see “Class Description” on page 7-1.

Class Names in Source Code

In source code, class names can be used in only two very different contexts. These contexts reflect the dual role of a class as a data type and as an object:

- The class name can be used as a type name for a kind of object. For example:

```
NSMatrix *anNSObject;
anNSObject = [[NSMatrix alloc] init];
```

Here an `NSObject` is statically typed to be an `NSMatrix`. The compiler will expect it to have the data structure of an `NSMatrix` instance and the instance methods defined and inherited by the `NSMatrix` class. Static typing enables the compiler to do better type checking and makes source code more self-documenting. See “Static Options” on page 9-22 for details.

Only instances can be statically typed; class objects cannot be, since they are not members of a class, but rather belong to the `Class` data type.

- As the receiver in a message expression, the class name refers to the class object. This usage was illustrated in several of the examples above. The class name can stand for the class object only as a message receiver. In any other context, you must ask the class object to reveal its `id` (by sending it a `class` message). The example below passes the `NSMatrix` class as an argument in an `isKindOfClass:` message.

```
if ( [anObject isKindOfClass:[NSMatrix class]] )
    . . .
```

It would have been illegal to simply use the name `NSMatrix` as the argument. The class name can only be a receiver.

If you do not know the class name at compile time, but have it as a string at run time, `objc_lookupClass()` will return the class object:

```
if ( [anObject isKindOfClass:objc_lookupClass(aBuffer)] )
    . . .
```

This function returns `nil` if the string passed is not a valid class name.

Class names compete in the same name space as variables and functions. A class and a global variable cannot have the same name. Class names are about the only names with global visibility in Objective C.

Defining a Class

Much of object-oriented programming consists of writing the code for new objects—defining new classes. In Objective C, classes are defined in two parts:

- An *interface* that declares the methods and instance variables of the class and names its superclass
- An *implementation* that actually defines the class (contains the code that implements its methods)

Although the compiler does not require it, the interface and implementation are usually separated into two different files. The interface file must be made available to anyone who uses the class. You generally would not want to distribute the implementation file that widely; users do not need source code for the implementation.

A single file can declare or implement more than one class. Nevertheless, it is customary to have a separate interface file for each class, if not also a separate implementation file. Keeping class interfaces separate better reflects their status as independent entities.

Interface and implementation files typically are named after the class. The implementation file has a `.m` suffix, indicating that it contains Objective C source code. The interface file can be assigned any other extension. Because it is included in other source files, the interface file usually has the `.h` suffix typical of header files. For example, the `NSMatrix` class would be declared in `NSMatrix.h` and defined in `NSMatrix.m`.

Separating an object's interface from its implementation fits well with the design of object-oriented programs. An object is a self-contained entity that can be viewed from the outside almost as a “black box.” Once you have determined how an object will interact with other elements in your program—that is, once you have declared its interface—you can freely alter its implementation without affecting any other part of the application.

The Interface

The declaration of a class interface begins with the compiler directive `@interface` and ends with the directive `@end`. (All Objective C directives to the compiler begin with “@”.)

```
@interface ClassName : ItsSuperclass
{
    instance variable declarations
}
method declarations
@end
```

The first line of the declaration presents the new class name and links it to its superclass. The superclass defines the position of the new class in the inheritance hierarchy, as discussed under “Inheritance” on page 8-9. If the colon and superclass name are omitted, the new class is declared as a root class, a rival to the `NSObject` class.

Following the class declaration, braces enclose declarations of *instance variables*, the data structures that will be part of each instance of the class. In OpenStep, all instance variables are private and are accessed by using accessor methods for getting and setting them.

Methods for the class are declared next, after the braces enclosing instance variables and before the end of the class declaration. The names of methods that can be used by class objects, *class methods*, are preceded by a plus sign:

```
+ alloc;
```

The methods that instances of a class can use, *instance methods*, are marked with a minus sign:

```
- display;
```

Although it is not a common practice, you can define a class method and an instance method with the same name. A method can also have the same name as an instance variable. This is more common, especially if the method returns the value in the variable.

Method return types are declared using the standard C syntax for casting one type to another. For example:

```
- (int)tag;
```

Argument types are declared in the same way:

```
- setTag:(int)anInt;
```

If a return or argument type is not explicitly declared, it is assumed to be the default type for methods and messages—an `id`. The `alloc`, `display`, and `setTag:` methods illustrated in the examples above all return `ids`.

When there is more than one argument, they are declared within the method name after the colons. Arguments break the name apart in the declaration, just as in a message. For example:

```
- (void)selectCellAtRow:(int)row column:(int)col;
- (BOOL)getRow:(int *)row column:(int *)col ofCell:(NSCell *)aCell;
```

Methods that take a variable number of arguments declare them using a comma and an ellipsis, just as a function would. For example:

```
- makeGroup:group, ...;
```

Importing the Interface

The interface file must be included in any source module that depends on the class interface—that includes any module that creates an instance of the class, sends a message to invoke a method declared for the class, or mentions an instance variable declared in the class. The interface is usually included with the `#import` directive:

```
#import "NSMatrix.h"
```

This directive is identical to `#include`, except that it makes sure that the same file is never included more than once. It is therefore preferred, and is used in place of `#include` in code examples throughout OpenStep documentation.

To reflect the fact that a class definition builds on the definitions of inherited classes, an interface file begins by importing the interface for its superclass:

```
#import "ItsSuperclass.h"

@interface ClassName : ItsSuperclass
{
    instance variable declarations
}
method declarations
@end
```

This convention means that every interface file includes, indirectly, the interface files for all inherited classes. When a source module imports a class interface, it gets interfaces for the entire inheritance hierarchy that the class is built upon.

Referring to Other Classes

An interface file declares a class and, by importing its superclass, implicitly contains declarations for all inherited classes, from `NSObject` on down through its superclass. If the interface mentions classes not in this hierarchy, it must import them explicitly—or, better, declare them with the `@class` directive:

```
@class NSMatrix, NSArray;
```

This directive simply informs the compiler that `NSMatrix` and `NSArray` are class names. It does not import their interface files.

An interface file mentions class names when it statically types instance variables, return values, and arguments. For example, the following declaration mentions the `NSArray` class.

```
- getCells:(NSArray *)theNSCells;
```

Since declarations like this simply use the class name as a type and do not depend on any details of the class interface (its methods and instance variables), the `@class` directive gives the compiler sufficient forewarning of what to expect. However, where the interface to a class is actually used (instances created, messages sent), the class interface must be imported. Typically, an interface file uses `@class` to declare classes, and the corresponding implementation file imports their interfaces (since it will need to create instances of those classes or send them messages).

The `@class` directive minimizes the amount of code seen by the compiler and linker, and is therefore the simplest way to give a forward declaration of a class name. Being simple, it avoids potential problems that may come with importing files that import still other files. For example, if one class declares a statically typed instance variable of another class, and their two interface files import each other, neither class may compile correctly.

The Role of the Interface

The purpose of the interface file is to declare the new class to other source modules (and to other programmers). It contains all the information they need to work with the class (programmers might also appreciate a little documentation).

- Through its list of method declarations, the interface file lets other modules know what messages can be sent to the class object and instances of the class. Every method that can be used outside the class definition is declared in the

interface file; methods that are internal to the class implementation can be omitted. For added clarity however, it may be a good idea to group these private methods together in a category defined in the class implementation file (see “Categories” on page 9-1).

- It also lets the compiler know what instance variables an object contains and programmers know what variables their subclasses will inherit. Although instance variables are most naturally viewed as a matter of the implementation of a class rather than its interface, they must nevertheless be declared in the interface file. This is because the compiler must be aware of the structure of an object where it is used, not just where it is defined. As a programmer, however, you can generally ignore the instance variables of the classes you use, except when defining a subclass.
- The interface file also tells users how the class is connected into the inheritance hierarchy and what other classes—inherited or simply referred to somewhere in the class—are needed.

The Implementation

The definition of a class is structured very much like its declaration. It begins with an `@implementation` directive and ends with `@end`:

```
@implementation ClassName : ItsSuperclass
{
    instance variable declarations
}
method definitions
@end
```

However, every implementation file must import its own interface. For example, `NSMatrix.m` imports `NSMatrix.h`. Because the implementation does not need to repeat any of the declarations it imports, it can safely omit the following:

- The name of the superclass
- The declarations of instance variables

This simplifies the implementation and devotes it mainly to method definitions:

```
#import "ClassName.h"

@implementation ClassName
method definitions
@end
```

Methods for a class are defined, like C functions, within a pair of braces. Before the braces, they are declared in the same manner as in the interface file, but without the semicolon. For example:

```
+ alloc
{
    . . .
}

- (int)tag
{
    . . .
}

- (void)setFrameOrigin:(NSPoint)newOrigin
{
    . . .
}
```

Methods that take a variable number of arguments handle them just as a functions would:

```
#import <stdarg.h>

- getGroup:group, ...
{
    va_list ap;
    va_start(ap, group);
    . . .
}
```

Referring to Instance Variables

Note – In OpenStep, all instance variables are private and are accessed by using accessor methods for getting and setting them.

By default, the definition of an instance method has all the instance variables of a potential receiving object within its scope. It can refer to them simply by name. Although the compiler creates the equivalent of C structures to store instance variables, the exact nature of the structure is hidden. You do not need either of the structure operators (‘.’ or ‘->’) to refer to an object’s data. For example, the following method definition refers to the receiver’s `tag` instance variable:

```
- setTag:(int)anInt
{
    tag = anInt;
    . . .
}
```

Neither the receiving object nor its `tag` instance variable is declared as an argument to this method, yet the instance variable falls within its scope. This simplification of method syntax is a significant shorthand in the writing of Objective C code.

The instance variables of the receiving object are not the only ones that you can refer to within the implementation of a class. You can refer to any instance variable of any object as long as the following two conditions are met:

- The instance variable must be within the scope of the class definition. Normally that means the instance variable must be one that the class declares or inherits. (Scope is discussed in more detail in “The Scope of Instance Variables” on page 8-28.)
- The compiler must know to what kind of object the instance variable belongs.

When the instance variable belongs to the receiver (as it does in the `setTag:` example above), this second condition is met automatically. The receiver’s type is implicit but clear—it is the very type that the class defines.

When the instance variable belongs to an object that is not the receiver, the object’s type must be made explicit to the compiler through static typing. In referring to the instance variable of a statically typed object, the structure pointer operator (`'->'`) is used.

Suppose, for example, that the `Sibling` class declares a statically typed object, `twin`, as an instance variable:

```
@interface Sibling : Object
{
    Sibling *twin;
    int gender;
    struct features *appearance;
}
```

As long as the instance variables of the statically typed object are within the scope of the class (as they are here because `twin` is typed to the same class), a `Sibling` method can set them directly:

```
- makeIdenticalTwin
{
    if ( !twin ) {
        twin = [[Sibling alloc] init];
        twin->gender = gender;
        twin->appearance = appearance;
    }
    return twin;
}
```

The Scope of Instance Variables

Although they are declared in the class interface, instance variables are more a matter of the way a class is implemented than of the way it is used. An object's interface lies in its methods, not in its internal data structures.

Often there is a one-to-one correspondence between a method and an instance variable, as in the following example:

```
- (int)tag
{
    return tag;
}
```

But this need not be the case. Some methods might return information not stored in instance variables, and some instance variables might store information that an object is unwilling to reveal.

As a class is revised from time to time, the choice of instance variables may change, even though the methods it declares remain the same. As long as messages are the vehicle for interacting with instances of the class, these changes will not really affect its interface.

To enforce the ability of an object to hide its data, the compiler limits the scope of instance variables—that is, limits their visibility within the program. But to provide flexibility, it also lets you explicitly set the scope at three different levels. (see Table 8-1 on page 8-29). Each level is marked by a compiler directive.

Table 8-1 Scope Levels for Instance Variables

Directive	Meaning
@private	The instance variable is accessible only within the class that declares it.
@protected	The instance variable is accessible within the class that declares it and within classes that inherit it.
@public	The instance variable is accessible everywhere.

This is illustrated in Figure 8-4.

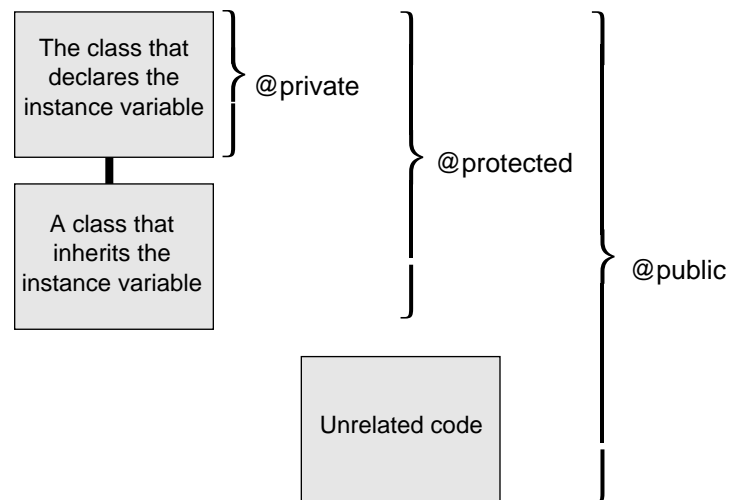


Figure 8-4 The Scope of Instance Variables

A directive applies to all the instance variables listed after it, up to the next directive or the end of the list. In the following example, the `age` and `evaluation` instance variables are private; `name`, `job`, and `wage` are protected; and `boss` is public.

```
@interface Worker : Object
{
    char *name;
    @private
    int age;
    char *evaluation;
    @protected
    id job;
    float wage;
    @public
    id boss;
}
```

By default, all unmarked instance variables (like `name` above) are `@protected`.

All instance variables that a class declares, no matter how they are marked, are within the scope of the class definition. For example, a class that declares a `job` instance variable, such as the `Worker` class shown in the example above, can refer to it in a method definition:

```
- promoteTo:newPosition
{
    id old = job;
    job = newPosition;
    return old;
}
```

Obviously, if a class could not access its own instance variables, the instance variables would be of no use whatsoever.

Normally, a class also has access to the instance variables it inherits. The ability to refer to an instance variable is usually inherited along with the variable. It makes sense for classes to have their entire data structures within their scope, especially if you think of a class definition as merely an elaboration of the classes it inherits from. The `promoteTo:` method illustrated above could just as well have been defined in any class that inherits the `job` instance variable from the `Worker` class.

However, there are reasons why you might want to restrict inheriting classes from accessing an instance variable:

- Once a subclass accesses an inherited instance variable, the class that declares the variable is tied to that part of its implementation. In later versions, it cannot eliminate the variable or alter the role it plays without inadvertently breaking the subclass.

- Moreover, if a subclass accesses an inherited instance variable and alters its value, it may inadvertently introduce bugs in the class that declares the variable, especially if the variable is involved in class-internal dependencies.

To limit an instance variable's scope to just the class that declares it, you must mark it `@private`.

At the other extreme, marking a variable `@public` makes it generally available, even outside of class definitions that inherit or declare the variable. Normally, to get information stored in an instance variable, other modules must send a message requesting it. However, a public instance variable can be accessed anywhere as if it were a field in a C structure.

```
Worker *ceo = [[Worker alloc] init];
ceo->boss = nil;
```

The object must be statically typed.

Marking instance variables `@public` defeats the ability of an object to hide its data. It runs counter to a fundamental principle of object-oriented programming—the encapsulation of data within objects where it is protected from view and inadvertent error. Public instance variables should therefore be avoided except in extraordinary cases.

How Messaging Works

In Objective C, messages are not bound to method implementations until run time. The compiler converts a message expression,

`[receiver message]`

into a call on a messaging function, `objc_msgSend()`. This function takes the receiver and the name of the method mentioned in the message—that is, the method selector—as its two principal arguments:

```
objc_msgSend(receiver, selector)
```

Any arguments passed in the message are also handed to `objc_msgSend()`:

```
objc_msgSend(receiver, selector, arg1, arg2, . . .)
```

The messaging function does everything necessary for dynamic binding:

- It first finds the procedure (method implementation) to which the selector refers. Since the same method can be implemented differently by different classes, the precise procedure that it finds depends on the class of the receiver.

- It then calls the procedure, passing it the receiving object (a pointer to its data), along with any arguments that were specified for the method.
- Finally, it passes on the return value of the procedure as its own return value.

Note – The compiler generates calls to the messaging function. You should never call it directly in the code you write.

The key to messaging lies in the structures that the compiler builds for each class and object. Every class structure includes these two essential elements:

- A pointer to the superclass
- A class *dispatch table*. This table has entries that associate method selectors with the class-specific addresses of the methods they identify. The selector for the `setFrameOrigin:` method is associated with the address of (the procedure that implements) `setFrameOrigin:`, the selector for the `display` method is associated with `display`'s address, and so on.

When a new object is created, memory for the object is allocated and its instance variables are initialized. First among the object's variables is a pointer to its class structure. This pointer, `ca`, gives the object access to its class and, through the class, to all the classes it inherits from.

These elements of class and object structure are illustrated in Figure 8-5 on page 8-33.

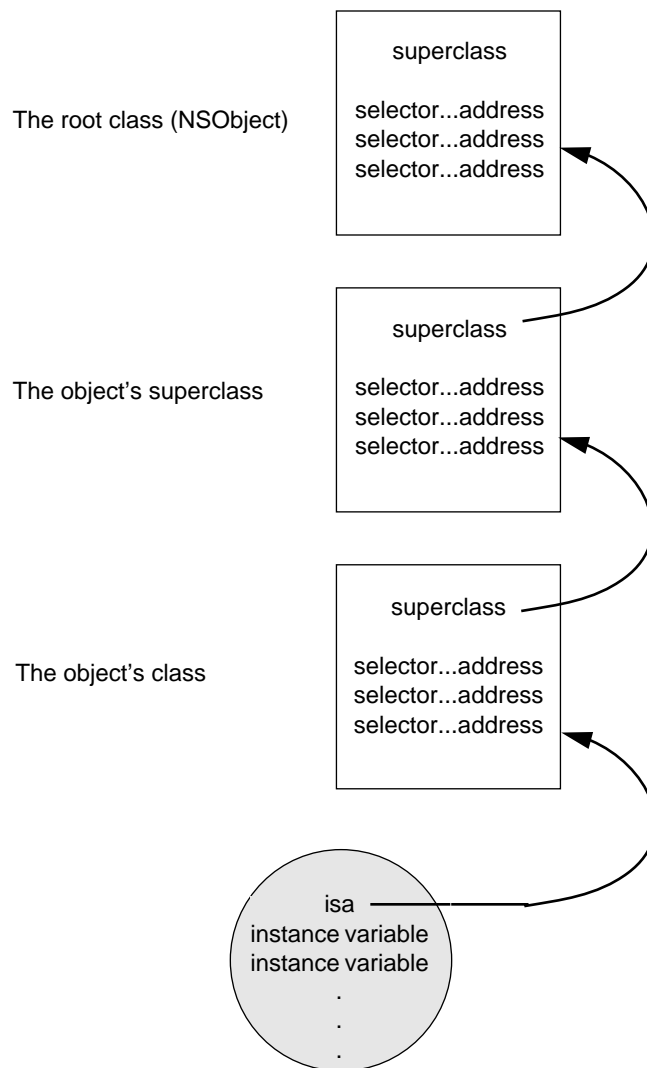


Figure 8-5 Messaging Framework

When a message is sent to an object, the messaging function follows the object's `isa` pointer to the class structure, where it looks up the method selector in the dispatch table. If it cannot find the selector there, `objc_msgSend()` follows the pointer to the superclass and tries to find the selector in its dispatch table.

Successive failures cause `objc_msgSend()` to climb the class hierarchy until it reaches the `NSObject` class. Once it locates the selector, it calls the method entered in the table and passes it the receiving object's data structure.

This is the way that method implementations are chosen at run time—or, in the jargon of object-oriented programming, that methods are dynamically bound to messages.

To speed the messaging process, the run-time system caches the selectors and addresses of methods as they are used. There is a separate cache for each class, and it can contain selectors for inherited methods as well as for methods defined in the class. Before searching the dispatch tables, the messaging routine first checks the cache of the receiving object's class on the theory that a method that was used once may likely be used again. If the method selector is in the cache, messaging is only slightly slower than a function call. Once a program has been running long enough to “warm up” its caches, almost all the messages it sends will find a cached method. Caches grow dynamically to accommodate new messages as the program runs.

Selectors

For efficiency, full ASCII names are not used as method selectors in compiled code. Instead, the compiler writes each method name into a table, then pairs the name with a unique identifier that will represent the method at run time. The run-time system makes sure each identifier is unique: No two selectors are the same, and all methods with the same name have the same selector. Compiled selectors are assigned to a special type, `SEL`, to distinguish them from other data. Valid selectors are never 0.

A compiled selector contains fields of coded information that aid run-time messaging. You should therefore let the system assign `SEL` identifiers to methods; it will not work to assign them arbitrarily yourself.

The `@selector()` directive lets Objective C source code refer to the compiled selector, rather than to the full method name. Here the selector for `setFrameOrigin:` is assigned to the `mover` variable:

```
SEL mover;  
mover = @selector(setFrameOrigin:);
```

It is most efficient to assign values to `SEL` variables at compile time with the `@selector()` directive. However, in some cases, a program may need to convert a character string to a selector at run time. This can be done with the `sel_getUid()` function:

```
mover = sel_getUid(aBuffer);
```

Conversion in the opposite direction is also possible. The `sel_getName()` function returns a method name for a selector:

```
char *method;  
method = sel_getName(mover);
```

Methods and Selectors

Compiled selectors identify method names, not method implementations. `NSView`'s `display` method, for example, will have the same selector as `display` methods defined in other classes. This is essential for polymorphism and dynamic binding; it lets you send the same message to receivers belonging to different classes. If there were one selector per method implementation, a message would be no different than a function call.

A class method and an instance method with the same name are assigned the same selector. However, because of their different domains, there is no confusion between the two. A class could define a `display` class method in addition to a `display` instance method.

Method Return and Argument Types

The messaging routine has access to method implementations only through selectors, so it treats all methods with the same selector alike. It discovers the return type of a method, and the data types of its arguments, from the selector. Therefore, except for messages sent to statically typed receivers, dynamic binding requires all implementations of identically named methods to have the same return type and the same argument types. (Statically typed receivers are an exception to this rule, since the compiler can learn about the method implementation from the class type.)

Although identically named class methods and instance methods are represented by the same selector, they can have different argument and return types.

Varying the Message at Run Time

The `perform:`, `perform:withObject:`, and `perform:withObject:withObject:` methods, defined in the `NSObject` class, take SEL identifiers as their initial arguments. All three methods map directly into the messaging function. For example,

```
[friend perform:@selector(gossipAbout:) withObject:aNeighbor];
```

is equivalent to:

```
[friend gossipAbout:aNeighbor];
```

These methods make it possible to vary a message at run time, just as it is possible to vary the object that receives the message. Variable names can be used in both halves of a message expression:

```
id helper = getTheReceiver();
SEL request = getTheSelector();
[helper perform:request];
```

In this example, the receiver (`helper`) is chosen at run time (by the fictitious `getTheReceiver()` function), and the method the receiver is asked to perform (`request`) is also determined at run time (by the equally fictitious `getTheSelector()` function).

Note - `perform:` and its companion methods return an `id`. If the method that is performed returns a different type, it should be cast to the proper type. (However, casting will not work for all types; the method should return a pointer or a type compatible with a pointer.)

The Target-Action Paradigm

In its treatment of user-interface controls, the OpenStep Application Kit makes good use of the ability to vary both the receiver and the message.

Controls are graphical devices that can be used to give instructions to an application. Most resemble real-world control devices such as buttons, switches, knobs, text fields, dials, menu items, and the like. In software, these devices stand between the application and the user. They interpret events coming from hardware devices like the keyboard and mouse and translate them into

application-specific instructions. For example, a button labeled “Find” would translate a mouse click into an instruction for the application to start searching for something.

The Application Kit defines a framework for creating control devices and defines a few “off-the-shelf” devices of its own. For example, the `NSButtonCell` class defines an object that you can assign to an `NSMatrix` and initialize with a size, a label, a picture, a font, and a keyboard alternative. When the user clicks the button (or uses the keyboard alternative), the `NSButtonCell` sends a message instructing the application to do something. To do this, an `NSButtonCell` must be initialized not just with an image, a size, and a label, but with directions on what message to send and to whom to send it. Accordingly, an `NSButtonCell` can be initialized for an *action message*, the method selector it should use in the message it sends, and a *target*, the object that should receive the message.

```
[myNSButtonCell setAction:@selector(reapTheWind)];  
[myNSButtonCell setTarget:anObject];
```

The `NSButtonCell` sends the message using `NSObject`'s `perform:withObject` method. All action messages take a single argument, the `id` of the control device sending the message.

If Objective C did not allow the message to be varied, all `NSButtonCells` would have to send the same message; the name of the method would be frozen in the `NSButtonCell` source code. Instead of simply implementing a mechanism for translating user actions into action messages, `NSButtonCells` and other controls would have to constrain the content of the message. This would make it difficult for any object to respond to more than one `NSButtonCell`. There would either have to be one target for each button, or the target object would have to discover which button the message came from and act accordingly. Each time you rearranged the user interface, you would also have to reimplement the method that responds to the action message. This would be an unnecessary complication that Objective C happily avoids.

Avoiding Messaging Errors

If an object receives a message to perform a method that is not in its repertoire, an error results. It is the same sort of error as calling a nonexistent function. But because messaging occurs at run time, the error often will not be evident until the program executes.

It is relatively easy to avoid this error when the message selector is constant and the class of the receiving object is known. As you are programming, you can check to be sure that the receiver is able to respond. If the receiver is statically typed, the compiler will check for you.

However, if the message selector or the class of the receiver varies, it may be necessary to postpone this check until run time. The `respondsToSelector:` method, defined in the `NSObject` protocol, determines whether a potential receiver can respond to a potential message. It takes the method selector as an argument, and returns whether the receiver has access to a method matching the selector:

```
if ( [anObject respondsToSelector:@selector(moveTo:)] )
    [anObject moveTo:0.0 :0.0];
else
    fprintf(stderr, "%s can't be moved\n", [anObject name]);
```

The `respondsToSelector:` test is especially important when sending messages to objects that you do not have control over at compile time. For example, if you write code that sends a message to an object represented by a variable that others can set, you should check to be sure the receiver implements a method that can respond to the message.

Note – An object can also arrange to have messages it receives forwarded to other objects, if it cannot respond to them directly itself. In that case, it will appear that the object cannot handle the message, even though it responds to it indirectly by assigning it to another object.

Hidden Arguments

When the messaging function finds the procedure that implements a method, it calls the procedure and passes it all the arguments in the message. It also passes the procedure two hidden arguments:

- The receiving object
- The selector for the method

These arguments give every method implementation explicit information about the two halves of the message expression that invoked it. They are said to be “hidden” because they are not declared in the source code that defines the method. They are inserted into the implementation when the code is compiled.

Although these arguments are not explicitly declared, source code can still refer to them just as it can refer to the receiving object's instance variables. A method refers to the receiving object as `self`, and to its own selector as `_cmd`. In the example below, `_cmd` refers to the selector for the `strange` method and `self` to the object that receives a `strange` message.

```
- strange
{
    id target = getTheReceiver();
    SEL action = getTheMethod();

    if ( target == self || action == _cmd )
        return nil;
    return [target perform:action];
}
```

Of the two arguments, `self` is the more useful. It is, in fact, the way the receiving object's instance variables are made available to the method definition.

“Messages to `self` and `super`” discusses `self` in more detail.

Messages to `self` and `super`

Objective C provides two terms that can be used within a method definition to refer to the object that performs the method—`self` and `super`.

Suppose, for example, that you define a `reposition` method that needs to change the coordinates of whatever object it acts on. It can invoke the `setFrameOrigin:` method to make the change. All it needs to do is send a `setFrameOrigin:` message to the very same object that the `reposition` message itself was sent to. When you are writing the `reposition` code, you can refer to that object as either `self` or `super`. The `reposition` method could read either:

```
- reposition
{
    . . .
    [self setFrameOrigin:someOrigin];
    . . .
}
```

or:

```
- reposition
{
    . . .
    [super moveTo:someX :someY];
    . . .
}
```

Here `self` and `super` both refer to the object receiving a `reposition` message, whatever object that may happen to be. The two terms are quite different, however. The term `self` is one of the hidden arguments that the messaging routine passes to every method; it is a local variable that can be used freely within a method implementation, just as the names of instance variables can be. The term `super` substitutes for `self` only as the receiver in a message expression. As receivers, the two terms differ principally in how they affect the messaging process:

- `self` searches for the method implementation in the usual manner, starting in the dispatch table of the receiving object's class. In the example above, it would begin with the class of the object receiving the `reposition` message.
- `super` starts the search for the method implementation in a very different place. It begins in the superclass of the class that defines the method where `super` appears. In the example above, it would begin with the superclass of the class where `reposition` is defined.

Wherever `super` receives a message, the compiler substitutes another messaging routine for `objc_msgSend()`. The substitute routine looks directly to the superclass of the defining class—that is, to the superclass of the class sending the message to `super`—rather than to the class of the object receiving the message.

An Example

The difference between `self` and `super` becomes clear in a hierarchy of three classes. Suppose, for example, that we create an object belonging to a class called `Low`. `Low`'s superclass is `Mid`; `Mid`'s superclass is `High`. All three classes define a method called `negotiate`, which they use for a variety of purposes. In addition, `Mid` defines an ambitious method called `makeLastingPeace`, which also has need of the `negotiate` method. This is illustrated in Figure 8-6 on page 8-41.

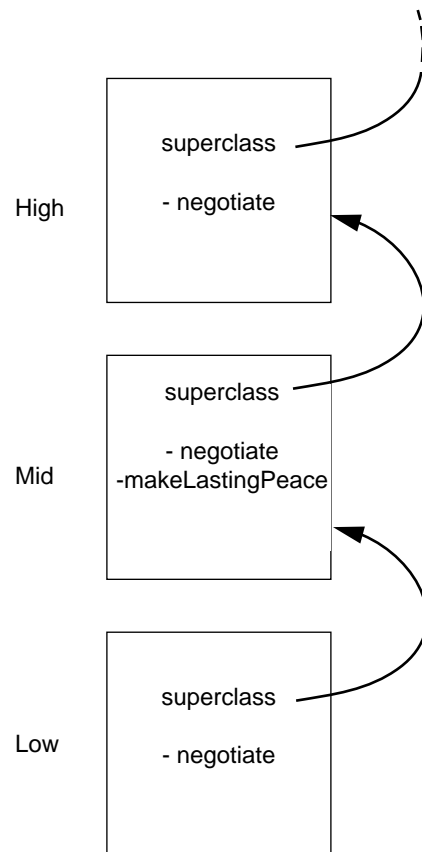


Figure 8-6 High, Mid, and Low

Now we send a message to our Low object to perform the `makeLastingPeace` method, and `makeLastingPeace`, in turn, sends a `negotiate` message to the same Low object. If source code calls this object `self`, the messaging routine will find the version of `negotiate` defined in Low, `self`'s class.

```
- makeLastingPeace
{
    [self negotiate];
    . . .
}
```

However, if source code calls this object `super`, the messaging routine will find the version of `negotiate` defined in `High`.

```
- makeLastingPeace
{
    [super negotiate];
    . . .
}
```

It ignores the receiving object's class (`Low`) and skips to the superclass of `Mid`, since `Mid` is where `makeLastingPeace` is defined. Neither message finds `Mid`'s version of `negotiate`.

As this example illustrates, `super` provides a way to bypass a method that overrides another method. Here it enabled `makeLastingPeace` to avoid the `Mid` version of `negotiate` that redefined the original `High` version.

Not being able to reach `Mid`'s version of `negotiate` may seem like a flaw, but, under the circumstances, it is right to avoid it:

- The author of the `Low` class intentionally overrode `Mid`'s version of `negotiate` so that instances of the `Low` class (and its subclasses) would invoke the redefined version of the method instead. The designer of `Low` did not want `Low` objects to perform the inherited method.
- In sending the message to `super`, the author of `Mid`'s `makeLastingPeace` method intentionally skipped over `Mid`'s version of `negotiate` (and over any versions that might be defined in classes like `Low` that inherit from `Mid`) to perform the version defined in the `High` class. `Mid`'s designer wanted to use the `High` version of `negotiate` and no other.

`Mid`'s version of `negotiate` could still be used, but it would take a direct message to a `Mid` instance to do it.

Using `super`

Messages to `super` allow method implementations to be distributed over more than one class. You can override an existing method to modify or add to it, and still incorporate the original method in the following modification:

```
- negotiate
{
    . . .
    return [super negotiate];
}
```

For some tasks, each class in the inheritance hierarchy can implement a method that does part of the job, and pass the message on to `super` for the rest. The `init` method, which initializes a newly allocated instance, and the `write:` method, which archives an object by writing it to a data stream, are designed to work this way. Each `write:` method has responsibility for writing the instance variables defined in its class. But before doing so, it sends a `write:` message to `super` to have the classes from which it inherits archive their instance variables. Each version of `write:` follows this same procedure, so classes write their instance variables in the order of inheritance:

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    . . .
    return self;
}
```

It is also possible to concentrate core functionality in one method defined in a superclass, and have subclasses incorporate the method through messages to `super`. For example, every class method that creates a new instance must allocate storage for the new object and initialize its `isa` pointer to the class structure. This is typically left to the `alloc` and `allocFromZone:` methods defined in the `NSObject` class. If another class overrides these methods for any reason (a rare case), it can still get the basic functionality by sending a message to `super`.

Redefining `self`

`super` is simply a flag to the compiler telling it where to begin searching for the method to perform; it is used only as the receiver of a message. But `self` is a variable name that can be used in any number of ways, even assigned a new value.

There is a tendency to do just that in definitions of class methods. Class methods are often concerned, not with the class object, but with instances of the class. For example, a method might combine allocation and initialization of an instance:

```
+ newTag:(int)anInt
{
    return [[self alloc] initWithTag:anInt];
}
```

In such a method, it is tempting to send messages to the instance and to call the instance `self`, just as in an instance method. But that would be an error. Both `self` and `super` refer to the receiving object—the object that gets a message telling it to perform the method. Inside an instance method, `self` refers to the instance; but inside a class method, `self` refers to the class object.

Before a class method can send a message telling `self` to perform an instance method, it must redefine `self` to be the instance:

```
+ newTag:(int)anInt andColor:(NSColor)aColor
{
    self = [[self alloc] initWithTag:anInt];
    [self setColor:aColor];
    return self;
}
```

The method shown above is a class method, so, initially, `self` refers to the class object. It is as the class object that `self` receives the `alloc` message. It is then redefined to be the instance that `alloc` returns and `initWithTag:` initializes. It is as the new instance that it receives the `setColor:` message.

To avoid confusion, it is usually better to use a variable other than `self` to refer to an instance inside a class method:

```
+ newTag:(int)anInt andColor:(NSColor)aColor
{
    id newInstance = [[self alloc] initWithTag:anInt];
    [newInstance setColor:aColor];
    return newInstance;
}
```

Note – In these examples, the class method sends messages (`initWithTag:` and `setColor:`) to initialize the instance. It does not assign a new value directly to an instance variable as an instance method might have done:

```
tag = anInt;
color = NS_REDCOLOR;
```

Only instance variables of the receiver can be directly set this way. Since the receiver for a class method (the class object) has no instance variables, this syntax cannot be used. However, if `newInstance` had been statically typed, something similar would have been possible:

```
newInstance->tag = anInt;
```

See “Referring to Instance Variables” on page 8-26 for more on when this syntax is permitted.

The Objective C Extensions



The preceding chapter has all you need to know about Objective C to define classes and design programs in the language. It covers basic Objective C syntax and explains the messaging process in detail.

Class definitions are at the heart of object-oriented programming, but they're not the only mechanism for structuring object definitions in Objective C. This chapter discusses two other ways of declaring methods and associating them with a class:

- Categories can compartmentalize a class definition or extend an existing one.
- Protocols declare methods that can be implemented by any class.

The chapter also explains how static typing works and takes up some lesser used features of Objective C, including ways to temporarily overcome its inherent dynamism.

Categories

You can add methods to a class by declaring them in an interface file under a category name and defining them in an implementation file under the same name. The category name indicates that the methods are additions to a class declared elsewhere, not a new class.

A category can be an alternative to a subclass. Rather than define a subclass to extend an existing class, through a category you can add methods to the class directly. For example, you could add categories to `NSMatrix` and other OpenStep classes. As in the case of a subclass, you do not need source code for the class you're extending.

The methods the category adds become part of the class type. For example, methods added to the `NSMatrix` class in a category will be among the methods the compiler will expect an `NSMatrix` instance to have in its repertoire. Methods added to the `NSMatrix` class in a subclass would not be included in the `NSMatrix` type. (This matters only for statically typed objects, since static typing is the only way the compiler can know an object's class.)

Category methods can do anything that methods defined in the class proper can do. At run time, there is no difference. The methods the category adds to the class are inherited by all the class's subclasses, just like other methods.

Adding to a Class

The declaration of a category interface looks very much like a class interface declaration—except the category name is listed within parentheses after the class name and the superclass is not mentioned. The category must import the interface file for the class it extends:

```
#import "ClassName.h"

@interface ClassName ( CategoryName )
method declarations
@end
```

The implementation, as usual, imports its own interface. Assuming that interface and implementation files are named after the category, a category implementation looks like this:

```
#import "CategoryName.h"

@implementation ClassName ( CategoryName )
method definitions
@end
```

A category cannot declare any new instance variables for the class; it includes only methods. However, all instance variables within the scope of the class are also within the scope of the category. That includes all instance variables declared by the class, even ones declared `@private`.

There's no limit to the number of categories that you can add to a class, but each category name must be different, and each should declare and define a different set of methods.

The methods added in a category can be used to extend the functionality of the class or override methods the class inherits. A category can also override methods declared in the class interface. However, it cannot reliably override methods declared in another category of the same class. A category is not a substitute for a subclass. It is best if categories do not attempt to redefine methods the class defines elsewhere; a class shouldn't define the same method more than once.

When a category overrides an inherited method, the new version can, as usual, incorporate the inherited version through a message to `super`. But there is no way for a category method to incorporate a method with the same name defined for the same class.

How Categories are Used

Categories can be used to extend classes defined by other implementors—for example, you can add methods to the classes defined in the OpenStep software kits. The added methods will be inherited by subclasses and will be indistinguishable at run time from the original methods of the class.

Categories can also be used to distribute the implementation of a new class into separate source files—for example, you could group the methods of a large class into several categories and put each category in a different file. When used like this, categories can benefit the development process in a number of ways:

- They provide a simple way of grouping related methods. Similar methods defined in different classes can be kept together in the same source file.
- They simplify the management of a large class when more than one developer is contributing to the class definition.
- They let you achieve some of the benefits of incremental compilation for a very large class.

- They can help improve locality of reference for commonly used methods.
- They enable you to configure a class differently for different applications, without having to maintain different versions of the same source code.

Categories are also used to declare informal protocols, as discussed under “Protocols” on page 8-48 below.

Categories of the Root Class

A category can add methods to any class, including the root `NSObject` class. Methods added to `NSObject` become available to all classes that are linked to your code. While this can be useful at times, it can also be quite dangerous. Although it may seem that the modifications the category makes are well understood and of limited impact, inheritance gives them a wide scope. You may be making unintended changes to unseen classes; you may not know all the consequences of what you are doing. Moreover, others who are unaware of your changes will not understand what they’re doing.

In addition, there are two other considerations to keep in mind when implementing methods for the root class:

- Messages to `super` are invalid (there is no superclass).
- Class objects can perform instance methods defined in the root class.

Normally, class objects can perform only class methods. But instance methods defined in the root class are a special case. They define an interface to the run-time system that all objects inherit. Class objects are full-fledged objects and need to share the same interface.

This feature means that you need to take into account the possibility that an instance method you define in a category of the `NSObject` class might be performed not only by instances but by class objects as well. For example, within the body of the method, `self` might mean a class object as well as an instance. See Chapter 7, “The `NSObject` Class,” for more information on class access to root instance methods.

Protocols

Class and category interfaces declare methods that are associated with a particular class—mainly methods that the class implements. Informal and formal protocols, on the other hand, declare methods not associated with a class, but which any class, and perhaps many classes, might implement.

A protocol is simply a list of method declarations, unattached to a class definition. For example, these methods that report user actions on the mouse could be gathered into a protocol:

```
- mouseDown:(NSEvent *)theEvent;  
- mouseDragged:(NSEvent *)theEvent;  
- mouseUp:(NSEvent *)theEvent;
```

Any class that wanted to respond to mouse events could adopt the protocol and implement its methods.

Protocols free method declarations from dependency on the class hierarchy, so they can be used in ways that classes and categories cannot. Protocols list methods that are (or may be) implemented somewhere, but the identity of the class that implements them is not of interest. What is of interest is whether or not a particular class conforms to the protocol—whether it has implementations of the methods the protocol declares. Thus objects can be grouped into types not just on the basis of similarities due to the fact that they inherit from the same class, but also on the basis of their similarity in conforming to the same protocol. Classes in unrelated branches of the inheritance hierarchy might be typed alike because they conform to the same protocol.

Protocols can play a significant role in object-oriented design, especially where a project is divided among many implementors or it incorporates objects developed in other projects. OpenStep software uses them heavily to support interprocess communication through Objective C messages.

However, an Objective C program does not need to use protocols. Unlike class definitions and message expressions, they are optional. Some OpenStep software kits use them; some do not. It all depends on the task at hand.

How Protocols are Used

Protocols are useful in at least three different situations:

- To declare methods that others are expected to implement

- To declare the interface to an object while concealing its class
- To capture similarities among classes that are not hierarchically related

The following sections discuss these situations and the roles protocols can play.

Methods for Others to Implement

If you know the class of an object, you can look at its interface declaration (and the interface declarations of the classes it inherits from) to find what messages it responds to. These declarations advertise the messages it can receive. Protocols provide a way for it to also advertise the messages it sends.

Communication works both ways; objects send messages as well as receive them. For example, an object might delegate responsibility for a certain operation to another object, or it may on occasion simply need to ask another object for information. In some cases, an object might be willing to notify other objects of its actions so that they can take whatever collateral measures might be required.

If you develop the class of the sender and the class of the receiver as part of the same project (or if someone else has supplied you with the receiver and its interface file), this communication is easily coordinated. The sender simply imports the interface file of the receiver. The imported file declares the method selectors the sender uses in the messages it sends.

However, if you develop an object that sends messages to objects that are not yet defined—objects that you are leaving for others to implement—you will not have the receiver's interface file. You need another way to declare the methods you use in messages but do not implement. A protocol serves this purpose. It informs the compiler about methods the class uses and also informs other implementors of the methods they need to define to have their objects work with yours.

Suppose, for example, that you develop an object that asks for the assistance of another object by sending it `helpOut:` and other messages. You provide an assistant instance variable to record the outlet for these messages and define a companion method to set the instance variable. This method lets other objects register themselves as potential recipients of your object's messages:

```
- setAssistant:anNSObject
{
    assistant = anNSObject;
```



```
        return self;
    }
```

Then, whenever a message is to be sent to the assistant, a check is made to be sure that the receiver implements a method that can respond:

```
- (BOOL)doWork
{
    . . .
    if ( [assistant respondsToSelector:@selector(helpOut:)] ) {
        [assistant helpOut:self];
        return YES;
    }
    return NO;
}
```

Since, at the time you write this code, you cannot know what kind of object might register itself as the assistant, you can only declare a protocol for the `helpOut:` method; you cannot import the interface file of the class that implements it.

Anonymous Objects

A protocol can also be used to declare the methods of an anonymous object, an object of unknown class. An anonymous object may represent a service or handle a limited set of functions, especially where only one object of its kind is needed. (Objects that play a fundamental role in defining an application's architecture and objects that you must initialize before using are not good candidates for anonymity.)

Objects cannot be anonymous to their developers, of course, but they can be anonymous when the developer supplies them to someone else. For example, an anonymous object might be part of a software kit or be located in a remote process:

- Someone who supplies a software kit or a suite of objects for others to use can include objects that are not identified by a class name or an interface file. Lacking the name and class interface, users have no way of creating instances of the class. Instead, the supplier must provide a ready-made instance. Typically, a method in another class returns a usable object:

```
id formatter = [receiver formattingService];
```

The object returned by the method is an object without a class identity, at least not one the supplier is willing to reveal. For it to be of any use at all, the supplier must be willing to identify at least some of the messages that it can respond to. This is done by associating the object with a list of methods declared in a protocol.

- It is possible to send Objective C messages to remote objects-objects in other applications. (“Remote Messaging” on page 9-15, discusses this possibility in more detail.)

Each application has its own structure, classes, and internal logic. But you do not need to know how another application works or what its components are to communicate with it. As an outsider, all you need to know is what messages you can send (the protocol) and where to send them (the receiver).

An application that publishes one of its objects as a potential receiver of remote messages must also publish a protocol declaring the methods the object will use to respond to those messages. It does not have to disclose anything else about the object. The sending application does not need to know the class of the object or use the class in its own design. All it needs is the protocol.

Protocols make anonymous objects possible. Without a protocol, there would be no way to declare an interface to an object without identifying its class.

Even though the supplier of an anonymous object will not reveal its class, the object itself will reveal it at run time. A class message will return the anonymous object's class. The class object can then be queried with the name and superclass methods. However, there's usually little point in discovering this extra information; the information in the protocol is sufficient.

Nonhierarchical Similarities

If more than one class implements a set of methods, those classes are often grouped under an abstract class that declares the methods they have in common. Each subclass may reimplement the methods in its own way, but the inheritance hierarchy and the common declaration in the abstract class captures the essential similarity between the subclasses.

However, sometimes it's not possible to group common methods in an abstract class. Classes that are unrelated in most respects might nevertheless need to implement some similar methods. This limited similarity may not justify a hierarchical relationship. For example, many different kinds of classes might implement methods to facilitate reference counting:

```
- setRefCount:(int)count;  
- (int)RefCount;  
- incrementCount;  
- decrementCount;
```

These methods could be grouped into a protocol and the similarity between implementing classes accounted for by noting that they all conform to the same protocol.

Objects can be typed by this similarity (the protocols they conform to), rather than by their class. For example, an `NSMatrix` must communicate with the objects that represent its cells. The `NSMatrix` could require each of these objects to be a kind of `NSCell` (a type based on class) and rely on the fact that all objects that inherit from the `NSCell` class will have the methods needed to respond to `NSMatrix` messages. Alternatively, the `NSMatrix` could require objects representing cells to have methods that can respond to a particular set of messages (a type based on protocol). In this case, the `NSMatrix` would not care what class a cell object belonged to, just that it implemented the methods.

Informal Protocols

The simplest way of declaring a protocol is to group the methods in a category declaration:

```
@interface NSObject ( RefCounting )  
- setRefCount:(int)count;  
- (int)RefCount;
```

```
- incrementCount;  
- decrementCount;  
@end
```

Informal protocols are typically declared as categories of the `NSObject` class, since that broadly associates the method names with any class that inherits from `NSObject`. Since all classes inherit from the root class, the methods are not restricted to any part of the inheritance hierarchy. (It would also be possible to declare an informal protocol as a category of another class to limit it to a certain branch of the inheritance hierarchy, but there is little reason to do so.)

When used to declare a protocol, a category interface does not have a corresponding implementation. Instead, classes that implement the protocol declare the methods again in their own interface files and define them along with other methods in their implementation files.

An informal protocol bends the rules of category declarations to list a group of methods but not associate them with any particular class or implementation.

Being informal, protocols declared in categories do not receive much language support. There is no type checking at compile time nor a check at run time to see whether an object conforms to the protocol. To get these benefits, you must use a formal protocol.

Formal Protocols

The Objective C language provides a way to formally declare a list of methods as a protocol. Formal protocols are supported by the language and the run-time system. For example, the compiler can check for types based on protocols, and objects can introspect at run time to report whether or not they conform to a protocol.

Formal protocols are declared with the `@protocol` directive:

```
@protocol ProtocolName  
method declarations  
@end
```

For example, the reference-counting protocol could be declared like this:

```
@protocol ReferenceCounting  
- setRefCount:(int)count;
```

```
- (int)refCount;  
- incrementCount;  
- decrementCount;  
@end
```

Unlike class names, protocol names do not have global visibility. They live in their own name space.

A class is said to adopt a formal protocol if it agrees to implement the methods the protocol declares. Class declarations list the names of adopted protocols within angle brackets after the superclass name:

```
@interface ClassName : ItsSuperclass < protocol list >
```

Categories adopt protocols in much the same way:

```
@interface ClassName ( CategoryName ) < protocol list >
```

Names in the protocol list are separated by commas.

A class or category that adopts a protocol must import the header file where the protocol is declared. The methods declared in the adopted protocol are not declared elsewhere in the class or category interface.

It is possible for a class to simply adopt protocols and declare no other methods. For example, this class declaration,

```
@interface Formatter : NSObject < Formatting, Prettifying >  
@end
```

adopts the `Formatting` and `Prettifying` protocols, but declares no instance variables or methods of its own.

A class or category that adopts a protocol is obligated to implement all the methods the protocol declares. The compiler will issue a warning if it does not. The `Formatter` class above would define all the methods declared in the two protocols it adopts, in addition to any it might have declared itself.

Adopting a protocol is similar in some ways to declaring a superclass. Both assign methods to the new class. The superclass declaration assigns it inherited methods; the protocol assigns it methods declared in the protocol list.

Protocol Objects

Just as classes are represented at run time by class objects and methods by selector codes, formal protocols are represented by a special data type—instances of the Protocol class. Source code that deals with a protocol (other than to use it in a type specification) must refer to the Protocol object.

In many ways, protocols are similar to class definitions. They both declare methods, and at run time they're both represented by objects—classes by class objects and protocols by Protocol objects. Like class objects, Protocol objects are created automatically from the definitions and declarations found in source code and are used by the run-time system. They're not allocated and initialized in program source code.

Source code can refer to a Protocol object using the `@protocol()` directive—the same directive that declares a protocol, except that here it has a set of trailing parentheses. The parentheses enclose the protocol name:

```
Protocol *counter = @protocol(ReferenceCounting);
```

This is the only way that source code can conjure up a Protocol object. Unlike a class name, a protocol name does not designate the object—except inside `@protocol()`.

The compiler creates a Protocol object for each protocol declaration it encounters, but only if the protocol is also:

- Adopted by a class, or
- Referred to somewhere in source code (using `@protocol()`).

Protocols that are declared but not used (except for type checking as described below) are not represented by Protocol objects.

Conforming to a Protocol

A class is said to conform to a formal protocol if it adopts the protocol or inherits from a class that adopts it. An instance of a class is said to conform to the same set of protocols its class conforms to.

Since a class must implement all the methods declared in the protocols it adopts, and those methods are inherited by its subclasses, saying that a class or an instance conforms to a protocol is tantamount to saying that it has in its repertoire all the methods that the protocol declares.

It is possible to check whether an object conforms to a protocol by sending it a `conformsTo:` message.

```
if ( [receiver conformsTo:@protocol(ReferenceCounting)] )
    [receiver incrementCount];
```

The `conformsTo:` test is very much like the `respondsTo:` test for a single method, except that it tests whether a protocol has been adopted (and presumably all the methods it declares implemented) rather than just whether one particular method has been implemented. Because it checks for a whole list of methods, `conformsTo:` can be more efficient than `respondsTo:`.

The `conformsTo:` test is also very much like the `isKindOfClass:` test, except that it tests for a type based on a protocol rather than a type based on the inheritance hierarchy.

Type Checking

Type declarations for objects can be extended to include formal protocols. Protocols thus offer the possibility of another level of type checking by the compiler, one that is more abstract since it is not tied to particular implementations.

In a type declaration, protocol names are listed between angle brackets after the type name:

```
- (id <Formatting>)formattingService;
id <ReferenceCounting, AutoFreeing> anObject;
```

Just as static typing permits the compiler to test for a type based on the class hierarchy, this syntax permits the compiler to test for a type based on conformance to a protocol.

For example, if `Formatter` is an abstract class, this declaration

```
Formatter *anObject;
```

groups all objects that inherit from `Formatter` into a type and permits the compiler to check assignments against that type.

Similarly, this declaration,

```
id <Formatting> anObject;
```

groups all objects that conform to the `Formatting` protocol into a type, regardless of their positions in the class hierarchy. The compiler can check to be sure that only objects that conform to the protocol are assigned to the type.

In each case, the type groups similar objects—either because they share a common inheritance, or because they converge on a common set of methods.

The two types can be combined in a single declaration:

```
Formatter <Formatting> *anObject;
```

Protocols cannot be used to type class objects. Only instances can be statically typed to a protocol, just as only instances can be statically typed to a class. (However, at run time, both classes and instances will respond to a `conformsTo: message`.)

Protocols within Protocols

One protocol can incorporate others using the same syntax that classes use to adopt a protocol:

```
@protocol ProtocolName < protocol list >
```

All the protocols listed between angle brackets are considered part of the `ProtocolName` protocol. For example, if the `Paging` protocol incorporates the `Formatting` protocol,

```
@protocol Paging < Formatting >
```

any object that conforms to the `Paging` protocol will also conform to `Formatting`. Type declarations

```
id <Paging> someObject;  
and conformsTo: messages  
if ( [anotherObject conformsTo:@protocol(Paging)] )  
    . . .
```

need mention only the `Paging` protocol to test for conformance to `Formatting` as well.

When a class adopts a protocol, it must implement the methods the protocol declares, as mentioned earlier. In addition, it must conform to any protocols the adopted protocol incorporates. If an incorporated protocol incorporates still other protocols, the class must also conform to them. A class can conform to an incorporated protocol by either:

Implementing the methods the protocol declares, or

Inheriting from a class that adopts the protocol and implements the methods.

Suppose, for example, that the `Pager` class adopts the `Paging` protocol. If `Pager` is a subclass of `NSObject`,

```
@interface Pager : NSObject < Paging >
```

it must implement all the `Paging` methods, including those declared in the incorporated `Formatting` protocol. It adopts the `Formatting` protocol along with `Paging`.

On the other hand, if `Pager` is a subclass of `Formatter` (a class that independently adopts the `Formatting` protocol),

```
@interface Pager : Formatter < Paging >
```

it must implement all the methods declared in the `Paging` protocol proper, but not those declared in `Formatting`. `Pager` inherits conformance to the `Formatting` protocol from `Formatter`.

Remote Messaging

Like most other programming languages, Objective C was initially designed for programs that are executed as a single process in a single address space.

Nevertheless, the object-oriented model, where communication takes place between relatively self-contained units through messages that are resolved at run-time, would seem well suited for interprocess communication as well. It is not hard to imagine Objective C messages between objects that reside in different address spaces (that is, in different tasks) or in different threads of execution of the same task.

For example, in a typical server-client interaction, the client task might send its requests to a designated object in the server, and the server might target specific client objects for the notifications and other information it sends.

Or imagine an interactive application that needs to do a good deal of computation to carry out a user command. It could simply put up an attention panel telling the user to wait while it was busy, or it could isolate the processing work in a subordinate task, leaving the main part of the application free to accept user input. Objects in the two tasks would communicate through Objective C messages.

Similarly, several separate processes could cooperate on the editing of a single document. There could be a different editing tool for each type of data in the document. One task might be in charge of presenting a unified user interface on-screen and of sorting out which user instructions were the responsibility of which editing tool. Each cooperating task could be written in Objective C, with Objective C messages being the vehicle of communication between the user interface and the tools and between one tool and another.

Distributed Objects

Remote messaging in Objective C requires a run-time system that can establish connections between objects in different address spaces, recognize when a message is intended for a remote address, and transfer data from one address space to another. It must also mediate between the separate schedules of the two tasks; it has to hold messages until their remote receivers are free to respond to them.

OpenStep includes a distributed objects architecture that is essentially this kind of extension to the run-time system. Using distributed objects, you can send Objective C messages to objects in other tasks or have messages executed in other threads of the same task. (When remote messages are sent between two threads of the same task, the threads are treated exactly like threads in different tasks.)

To send a remote message, an application must first establish a connection with the remote receiver. Establishing the connection gives the application a proxy for the remote object in its own address space. It then communicates with the remote object through the proxy. The proxy assumes the identity of the remote object; it has no identity of its own. The application is able to regard the proxy as if it were the remote object; for most purposes, it is the remote object.

Remote messaging is diagrammed in Figure 9-1, where object A communicates with object B through a proxy, and messages for B wait in a queue until B is ready to respond to them:

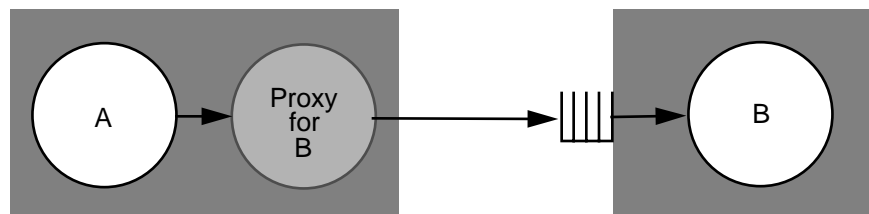


Figure 9-1 Remote Messages

The sender and receiver are in different tasks and are scheduled independently of each other. So there is no guarantee that the receiver will be free to accept a message when the sender is ready to send it. Therefore, arriving messages are placed in a queue and retrieved at the convenience of the receiving application.

A proxy does not act on behalf of the remote object or need access to its class. It is not a copy of the object, but a lightweight substitute for it. In a sense, it's transparent; it simply passes the messages it receives on to the remote receiver and manages the interprocess communication. Its main function is to provide a local address for an object that would not otherwise have one.

A remote receiver is typically anonymous. Its class is hidden inside the remote application. The sending application does not need to know how that application is designed or what classes it uses. It does not need to use the same classes itself. All it needs to know is what messages the remote object responds to.

Because of this, an object that's designated to receive remote messages typically advertises its interface in a formal protocol. Both the sending and the receiving application declare the protocol—they both import the same protocol declaration. The receiving application declares it because the remote object must conform to the protocol. The sending application declares it to inform the compiler about the messages it sends and because it may use the `conformsTo:` method and the `@protocol()` directive to test the remote receiver. The sending application does not have to implement any of the methods in the protocol; it declares the protocol only because it initiates messages to the remote receiver.

The distributed objects architecture, including the `NSProxy` and `NSConnection` classes, is documented in *OpenStep Programming Reference*.

Language Support

Remote messaging raises not only a number of intriguing possibilities for program design, it also raises some interesting issues for the Objective C language. Most of the issues are related to the efficiency of remote messaging and the degree of separation that the two tasks should maintain while they're communicating with each other.

So that programmers can give explicit instructions about the intent of a remote message, Objective C defines five type qualifiers that can be used when declaring methods inside a formal protocol:

`oneway`

`in`

`out`

`inout`

`bycopy`

These modifiers are restricted to formal protocols; they cannot be used inside class and category declarations. However, if a class or category adopts a protocol, its implementation of the protocol methods can use the same modifiers that are used to declare the methods.

The following sections explain how these five modifiers are used.

Synchronous and Asynchronous Messages

Consider first a method with just a simple return value:

```
- (BOOL) canDance;
```

When a `canDance` message is sent to a receiver in the same application, the method is invoked and the return value provided directly to the sender. But when the receiver is in a remote application, two underlying messages are required—one message to get the remote object to invoke the method, and the other message to send back the result of the remote calculation. This is illustrated in the figure below:

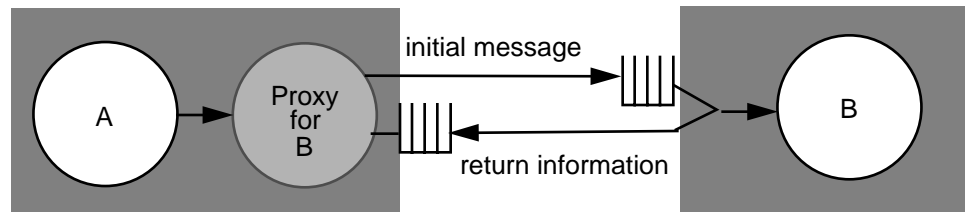


Figure 9-2 Round-Trip Message

Most remote messages will be, at bottom, two-way (or “round trip”) remote procedure calls (RPCs) like this one. The sending application waits for the receiving application to invoke the method, complete its processing, and send back an indication that it has finished, along with any return information requested. Waiting for the receiver to finish, even if no information is returned, has the advantage of coordinating the two communicating applications, of keeping them both “in sync.” For this reason, round-trip messages are often called synchronous. Synchronous messages are the default.

However, it is not always necessary or a good idea to wait for a reply. Sometimes it is sufficient simply to dispatch the remote message and return, allowing the receiver to get to the task when it will. In the meantime, the sender can go on to other things. Objective C provides a return type modifier, `oneway`, to indicate that a method is used only for asynchronous messages:

```
- (oneway void)waltzAtWill;
```

Although `oneway` is a type qualifier (like `const`) and can be used in combination with a specific type name, such as `oneway float` or `oneway id`, the only such combination that makes any sense is `oneway void`. An asynchronous message cannot have a valid return value.

Pointer Arguments

Next, consider methods that take pointer arguments. A pointer can be used to pass information to the receiver by reference. When invoked, the method looks at what is stored in the address it has passed.

```
- setTune:(struct tune *)aSong
{
    tune = *aSong;
    . . .
}
```

```
}

```

The same sort of argument can also be used to return information by reference. The method uses the pointer to find where it should place information requested in the message.

```
- getTune:(struct tune *)theSong
{
    . . .
    *theSong = tune;
}
```

The way the pointer is used makes a difference in how the remote message is carried out. In neither case can the pointer simply be passed to the remote object unchanged; it points to a memory location in the sender's address space and would not be meaningful in the address space of the remote receiver. The run-time system for remote messaging must make some adjustments behind the scenes.

If the argument is used to pass information by reference, the run-time system must dereference the pointer, ship the value it points to over to the remote application, store the value in an address local to that application, and pass that address to the remote receiver.

If, on the other hand, the pointer is used to return information by reference, the value it points to does not have to be sent to the other application. Instead, a value from the other application must be sent back and written into the location indicated by the pointer.

In the one case, information is passed on the first leg of the round trip. In the other case, information is returned on the second leg of the round trip. Because these cases result in very different actions on the part of the run-time system for remote messaging, Objective C provides type modifiers that can clarify the programmer's intention:

- The type modifier `in` indicates that information is being passed in a message:

```
- setTune:(in struct tune *)aSong;
```

- The modifier `out` indicates that an argument is being used to return information by reference:

```
- getTune:(out struct tune *)theSong;
```

- A third modifier, `inout`, indicates that an argument is used both to provide information and to get information back:

```
- adjustTune:(inout struct tune *)aSong;
```

The OpenStep distributed objects system takes `inout` to be the default modifier for all pointer arguments except those declared `const`, for which `in` is the default. `inout` is the safest assumption, but also the most time-consuming since it requires passing information in both directions. The only modifier that makes sense for arguments passed by value (nonpointers) is `in`. While `in` can be used with any kind of argument, `out` and `inout` make sense only for pointers.

In C, pointers are sometimes used to represent composite values. For example, a string is represented as a character pointer (`char *`). Although in notation and implementation there is a level of indirection here, in concept there is not. Conceptually, a string is an entity in and of itself, not a pointer to something else.

In cases like this, the distributed objects system automatically dereferences the pointer and passes whatever it points to as if by value. Therefore, the `out` and `inout` modifiers make no sense with simple character pointers. It takes an additional level of indirection in a remote message to pass or return a string by reference:

```
- getTuneTitle:(out char **)theTitle;
```

The same is true of objects:

```
- adjustMatrix:(inout Matrix **)theMatrix;
```

These conventions are enforced at run time, not by the compiler.

Proxies and Copies

Finally, consider a method that takes an object as an argument:

```
- danceWith:aPartner;
```

A `danceWith:` message passes an object id to the receiver. If the sender and receiver are in the same application, they would both be able to refer to the same `aPartner` object.

This is true even if the receiver is in a remote application, except that the receiver will need to refer to the object through a proxy (since the object is not in its address space). The pointer that `danceWith:` delivers to a remote

receiver is actually a pointer to the proxy. Messages sent to the proxy would be passed across the connection to the real object and any return information would be passed back to the remote application.

There are times when proxies may be unnecessarily inefficient, when it's better to send a copy of the object to the remote process so that it can interact with it directly in its own address space. To give programmers a way to indicate that this is intended, Objective C provides a `bycopy` type modifier:

```
- danceWith:(bycopy id)aClone;
```

`bycopy` can also be used for return values:

```
- (bycopy)dancer;
```

It can similarly be used with `out` to indicate that an object returned by reference should be copied rather than delivered in the form of a proxy:

```
- getDancer:(bycopy out id *)theDancer;
```

The only type that it makes sense for `bycopy` to modify is an object, whether dynamically typed `id` or statically typed by a class name.

When a copy of an object is passed to another application, it cannot be anonymous. The application that receives the object must have the class of the object loaded in its address space.

Static Options

Objective C objects are dynamic entities. As many decisions about them as possible are pushed from compile time to run time:

The memory for objects is dynamically allocated at run time by class methods that create new instances.

Objects are dynamically typed. In source code (at compile time), any object can be of type `id` no matter what its class. The exact class of an `id` variable (and therefore its particular methods and data structure) is not determined until the program is running.

Messages and methods are dynamically bound, as described under “How Messaging Works” on page 8-32 in the previous chapter. A run-time procedure matches the method selector in the message to a method implementation that “belongs to” the receiver.

These features give object-oriented programs a great deal of flexibility and power, but there is a price to pay. Messages are somewhat slower than function calls, for example, and the compiler cannot check the exact types (classes) of id variables.

To permit better compile-time type checking, and to make code more self-documenting, Objective C allows objects to be statically typed with a class name rather than generically typed as id.

Static Typing

If a pointer to a class name is used in place of id in an object declaration,

```
NSMatrix *thisNSObject;
```

the compiler restricts the declared variable to be either an instance of the class named in the declaration or an instance of a class that inherits from the named class. In the example above, this NSObject can only be an NSMatrix of some kind.

Statically typed objects have the same internal data structures as objects declared to be ids. The type does not affect the object; it affects only the amount of information given to the compiler about the object and the amount of information available to those reading the source code.

Static typing also does not affect how the object is treated at run time. Statically typed objects are dynamically allocated by the same class methods that create instances of type id. If Mosaic is a subclass of NSMatrix, the following code would still produce an object with all the instance variables of a Mosaic, not just those of an NSMatrix:

```
NSMatrix *thisNSObject = [[Mosaic alloc] init];
```

Messages sent to statically typed objects are dynamically bound, just as objects typed id are. The exact type of a statically typed receiver is still determined at run time as part of the messaging process. A display message sent to

```
thisNSObject  
[thisObject display];
```

will perform the version of the method defined in the Mosaic class, not its NSMatrix superclass.

By giving the compiler more information about an object, static typing opens up possibilities that are absent for objects typed id:

In certain situations, it allows for compile-time type checking.

It can free objects from the restriction that identically named methods must have identical return and argument types.

It permits you to use the structure pointer operator to directly access an object's instance variables.

The first two topics are discussed in the sections below. The third was covered in the previous chapter under “Defining a Class” on page 8-21.

Type Checking

With the additional information provided by static typing, the compiler can deliver better type-checking services in two situations:

- When a message is sent to a statically typed receiver, the compiler can check to be sure that the receiver can respond. A warning is issued if the receiver does not have access to the method named in the message.
- When a statically typed object is assigned to a statically typed variable, the compiler can check to be sure that the types are compatible. A warning is issued if they are not.

An assignment can be made without warning provided the class of the object being assigned is identical to, or inherits from, the class of the variable receiving the assignment. This is illustrated in the example below.

```
NSView      *aNSView;  
NSMatrix   *aNSMatrix;  
  
aNSMatrix = [[NSMatrix alloc] init];  
aNSView = aNSMatrix;
```

Here an `NSMatrix` can be assigned to an `NSView` because an `NSMatrix` is a kind of `NSView`—the `NSMatrix` class inherits from `NSView`. However, if the roles of the two variables are reversed and an `NSView` is assigned to an `NSMatrix`, the compiler will generate a warning; not every `NSView` is a `NSMatrix`. (For reference, Figure 8-3 in the previous chapter shows a portion of the class hierarchy including `NSView` and `NSMatrix`.)

There is no check when the expression on either side of the assignment operator is an id. A statically typed object can be freely assigned to an id, or an id to a statically typed object. Because methods like `alloc` and `init` return ids, the compiler does not check to be sure that a compatible object is returned to a statically typed variable. The following code is error-prone, but is allowed nonetheless:

```
NSMatrix *anNSMatrix;  
anNSMatrix = [[NSWindow alloc] init];
```

Note – This is consistent with the implementation of `void *` (pointer to void) in ANSI C. Just as `void *` is a generic pointer that eliminates the need for coercion in assignments between pointers, `id` is a generic pointer to objects that eliminates the need for coercion to a particular class in assignments between objects. For the purpose of type checking however, if a variable of type `id` is protocol qualified—that is, `id<myProtocol> myVar`—the compiler treats `myVar` as if it were statically typed, and issues warnings if a message sent to `myVar` is not declared in `myProtocol`.

Return and Argument Types

In general, methods that share the same selector (the same name) must also share the same return and argument types. This constraint is imposed by dynamic binding. Because the class of a message receiver, and therefore class-specific details about the method it's asked to perform, cannot be known at compile time, the compiler must treat all methods with the same name alike. When it prepares information on method return and argument types for the run-time system, it creates just one method description for each method selector.

However, when a message is sent to a statically typed object, the class of the receiver is known by the compiler. The compiler has access to class-specific information about the methods. Therefore, the message is freed from the restrictions on its return and argument types.

Static Typing to an Inherited Class

An instance can be statically typed to its own class or to any class that it inherits from. All instances, for example, can be statically typed as `NSObject`s.

However, the compiler understands the class of a statically typed object only from the class name in the type designation, and it does its type checking accordingly. Typing an instance to an inherited class can therefore result in discrepancies between what the compiler thinks would happen at run time and what will actually happen.

For example, if you statically type an `NSMatrix` instance as an `NSView`,

```
NSView *myNSMatrix = [[NSMatrix alloc] init];
```

the compiler will treat it as an `NSView`. If you send the object a message to perform an `NSMatrix` method,

```
id cell = [myNSMatrix selectedCell];
```

the compiler will complain. The `selectedCell` method is defined in the `NSMatrix` class, not in `NSView`.

However, if you send it a message to perform a method that the `NSView` class knows about,

```
[myNSMatrix display];
```

the compiler will not complain, even though `NSMatrix` overrides the method. At run time, `NSMatrix`'s version of the method will be performed.

Similarly, suppose that the `Upper` class declares a `worry` method that returns a `double`,

```
- (double)worry;
```

and the `Middle` subclass of `Upper` overrides the method and declares a new return type:

```
- (int)worry;
```

If an instance is statically typed to the `Upper` class, the compiler will think that its `worry` method returns a `double`, and if an instance is typed to the `Middle` class, it will think that `worry` returns an `int`. Errors will obviously result if a `Middle` instance is typed to the `Upper` class. The compiler will inform the run-time system that a `worry` message sent to the object will return a `double`, but at run time it will actually return an `int` and generate an error.

Static typing can free identically named methods from the restriction that they must have identical return and argument types, but it can do so reliably only if the methods are declared in different branches of the class hierarchy.

Getting a Method Address

The only way to circumvent dynamic binding is to get the address of a method and call it directly as if it were a function. This might be appropriate on the rare occasions when a particular method will be performed many times in succession and you want to avoid the overhead of messaging each time the method is performed.

With a method defined in the `NSObject` class, `methodForSelector:`, you can ask for a pointer to the procedure that implements a method, then use the pointer to call the procedure. The pointer that `methodForSelector:` returns must be carefully cast to the proper function type. Both return and argument types should be included in the cast.

The example below shows how the procedure that implements the `setTag:` method might be called:

```
id (*setter)(id, SEL, int);
int i;

setter = (id (*)(id, SEL, int))[target
methodForSelector:@selector(setTag:)];
for ( i = 0; i < 1000, i++ )
    setter(targetList[i], @selector(setTag:), i);
```

The first two arguments passed to the procedure are the receiving object (`self`) and the method selector (`_cmd`). These arguments are hidden in method syntax but must be made explicit when the method is called as a function.

Using `methodForSelector:` to circumvent dynamic binding saves most of the time required by messaging. However, the savings will be significant only where a particular message will be repeated many times, as in the for loop shown above.

Note that `methodForSelector:` is provided by the run-time system; it is not a feature of the Objective C language itself.

Getting an Object Data Structure

A fundamental tenet of object-oriented programming is that the data structure of an object is private to the object. Information stored there can be accessed only through messages sent to the object. However, there is a way to strip an object data structure of its "objectness" and treat it like any other C structure. This makes all the object's instance variables publicly available.

When given a class name as an argument, the `@defs()` directive produces the declaration list for an instance of the class. This list is useful only in declaring structures, so `@defs()` can appear only in the body of a structure declaration. This code, for example, declares a structure that would be identical to the template for an instance of the `Worker` class:

```
struct workerDef {
    @defs(Worker)
} *public;
```

Here `public` is declared as a pointer to a structure that is essentially indistinguishable from a `Worker` instance. With a little help from a type cast, a `Worker` id can be assigned to the pointer. The object's instance variables can then be accessed publicly through the pointer:

```
id aWorker;
aWorker = [[Worker alloc] init];

public = (struct workerDef *)aWorker;
public->boss = nil;
```

This technique of turning an object into a structure makes all of its instance variables public, no matter whether they were declared `@private`, `@protected`, or `@public`.

Objects generally are not designed with the expectation that they will be turned into C structures. You may want to use `@defs()` for classes you define entirely yourself, but it should not be applied to classes found in a library or to classes you define that inherit from library classes.

Type *Encoding

To assist the run-time system, the compiler encodes the return and argument types for each method in a character string and associates the string with the method selector. The coding scheme it uses might also be of use in other contexts and so is made publicly available with the `@encode()` directive. When given a type specification, `@encode()` returns a string encoding that type. The type can be a basic type such as an int, a pointer, a tagged structure or union, or a class name—anything, in fact, that can be used as an argument to the C `sizeof()` operator.

```
char *buf1 = @encode(int **);
char *buf2 = @encode(struct key);
char *buf3 = @encode(NSMatrix);
```

Table 9-1 lists the type codes. Note that many of them overlap with the codes used in writing to a typed stream. However, there are codes listed here that you cannot use when writing to a typed stream and there are codes that you may want to use when writing to a typed stream that are not generated by `@encode()`. (See *OpenStep Programming Reference* for information on typed streams.)

Table 9-1 Type Codes

Code	Meaning
c	A char
i	An int
s	A short
l	A long
C	An unsigned char
I	An unsigned int
S	An unsigned short
L	An unsigned long
f	A float
d	A double
v	A void

Table 9-1 Type Codes

Code	Meaning
*	A character string (char *)
@	An object (whether statically typed or typed id)
#	A class object (Class)
:	A method selector (SEL)
[...]	An array
{...}	A structure
(...)	A union
<i>bnum</i>	A bitfield of <i>num</i> bits
<i>^type</i>	A pointer to <i>type</i>
?	An unknown type

The type specification for an array is enclosed within square brackets; the number of elements in the array is specified immediately after the open bracket, before the array type. For example, an array of 12 pointers to floats would be encoded as:

```
[12^f]
```

Structures are specified within braces, and unions within parentheses. The structure tag is listed first, followed by an equal sign and the codes for the fields of the structure listed in sequence. For example, this structure,

```
typedef struct example {
    id anObject;
    char *aString;
    int anInt;
} Example;
```

would be encoded like this:

```
{example=@*i}
```

The same encoding results whether the defined type name (Example) or the structure tag (example) is passed to `@encode()`. The encoding for a structure pointer carries the same amount of information about the structure's fields:

```
^{example=@*i}
```

However, another level of indirection removes the internal type specification:

```
^^{example}
```

Objects are treated like structures. For example, passing the `NSObject` class name to `@encode()` yields this encoding:

```
{Object=#}
```

The `NSObject` class declares just one instance variable, `isa`, of type `Class`.

Although the `@encode()` directive does not return them, the run-time system also uses these additional encodings for type qualifiers when they are used to declare methods in a protocol:

Table 9-2 Additional Encodings

Code	Meaning
r	const
n	in
N	inout
o	out
O	bycopy
V	oneway

Debugging an OpenStep Application



The SPARCworks Debugger is an interactive, window-based, source code and machine-instruction level debugging tool. It provides dynamic analysis for observing run-time program behavior. The Debugger gives you complete control of the dynamic execution of a program, including the collection of performance data.

The Debugger provides the same functionality as `dbx`, the command-line debugging tool, and you can enter `dbx` commands in the Command Pane of the Debugger base window.

To debug an OpenStep application, click on the Debug button in the project window for the application in Project Builder. If the project has not been built yet, it is built first. If the project builds successfully, then the application is run in debug mode and the SPARCworks Debugger starts up. See the SPARCworks manual *Debugging a Program* for details on using the Debugger windows.

Note – If the project has already been built, you can Alt-click on the Debug button to run the application under the Debugger.

Debugger Objective C Support

Release 3.1 of the SPARCworks Debugger includes support for Objective C applications, such as those developed using OpenStep.

Dynamic Types

In Objective C an object pointer has two types:

- its static type, which is defined in the source code
- its dynamic type, which is known at run-time

The Debugger can provide information about the dynamic type of an object when you use the `print`, `display`, `inspect`, and `whatis` commands with the `-d` option, or when you have set the dbx customization variable `output_dynamic_type` to `on`. If you use the `+d` option, the commands will use the static type.

It is recommended that you put `dbxenv output_dynamic_type on` in your `~/.dbxrc` file when debugging Objective C programs.

Finding Methods and Using Method Names in Non-expression Commands

The following are non-expression commands:

```
stop in
funcs
whatis
list
edit
```

Use the `funcs` command (with a regular expression) to find methods and functions that the Debugger knows about and to print them in a format the Debugger accepts. Use the `dbx` command `help funcs` for more information on the `funcs` command.

If the process is active, the Debugger uses the run-time system to look up a method, otherwise it uses static information (stabs).

Setting Breakpoints

The Debugger accepts the following variations of the `stop` command for setting breakpoints in Objective C methods:

```
stop in -[Test ival:second:]
```

```
stop in +[Test alloc]
stop in [Test ival:second:]
stop in [obj ival:second:]// through an object (only if active
                           process)
stop in ``ival:second:    // searches all classes for ival:second:
stop in ival:second:      // only if dbxenv scope_look_aside is `on'
stop inmethod ival:second:// stops in all methods with that name
```

If the process is not active, use the following syntax for category methods:

```
stop in -[Test(Cat1) catmethod:second:]
```

Calling Objective C Methods

All Objective C instance methods must be called through an object. The following are some valid variations of calling Objective C methods:

```
call [obj ival: 30]          // calling instance method with parameter
call [self ival: 30]        // use self if stopped inside a class
call [Test alloc]           // calling class method
```

Recovering from a Run-time System Crash

The Debugger calls the Objective C run-time system to look up methods and, if `output_dynamic_type` is on, to find the dynamic type of an object. In some cases this can cause a crash of the run-time system. The Debugger can usually recover if you use the `pop` command. Use the `where` command and then the `pop` command to unwind frames from the stack. You can also use the `kill` command to return to a previous Debugger level.

Sample .dbxrc File

Your `~/ .dbxrc` file is read automatically if it exists when the Debugger starts up. The following is a sample `.dbxrc` file for debugging Objective C applications. This file is located in `/usr/openstep/etc`. To have the Debugger read this file when it starts up, add `source /usr/openstep/etc/.dbxrc` to your `.dbxrc` file.

```
#####
#                               Objective C settings                               #
#####

language objc
dbxenv scope_look_aside on // sets the dbx customization variable
                           scope_look_aside to on (find static
                           symbols even when not in scope)
dbxenv output_dynamic_type on // sets the dbx customization variable
                              output_dynamic_type to on (display,
                              inspect, print, whatis commands use
                              dynamic type of object)

# do
#   call objc_enableMessageSendDebug(1)
# to enable the tracing of messages in objc_msgSend. This tracing is
# very fast and flexible. The above command will echo back all the
# info you need to use this feature.

function objchelp
{
    echo "Add 'source /usr/openstep/etc/.dbxrc' to your ~/.dbxrc
file"
    echo " to define helpful Objective C functions, aliases and
buttons."
    echo " Look at this file in an editor to see what it contains."
    echo "For more help, enter:"
    echo "     help          general dbx help"
    echo "     help ObjC      more Objective C help"
    echo "     help FAQ        dbx - gdb correspondences, and other
information"
}

function memon # stop if an object is freed too many times. VERY
SLOW!!
{
```

```
        call [NSAutoreleasePool enableDoubleReleaseCheck: 1]
        stop in _NSAutoreleaseInconsistency
        status
    }

function memoff
{
    call [NSAutoreleasePool enableDoubleReleaseCheck: 0]
}

function defbrks # breakpoints that catch errors
{
    language objc
    stop in abort
    stop in -[NSObject doesNotRecognizeSelector:]
    stop in +[NSAssertionHandler currentHandler]
    stop in -[NSAssertionHandler
handleFailureInMethod:object:file:lineNumber:description:]
    stop in -[NSAssertionHandler
handleFailureInFunction:file:lineNumber:description:]
    stop in -[NSException raise]
    stop in DPSTefaultErrorProc
    stop in DPSCantHappen
    stop in _exit
}

function morebrks # other helpful places to breakpoint
{
    stop in NSLog
    stop in _XErrorHandler
    stop in -[Zombie forward::]
}

function allbrks # set breakpoints to catch errors
```

```
{
    defbrks
    morebrks
}

function pselfvar
{
    print self->${1}
}

function pdesc
{
    print [[$* description] cString]
}

function pnsstring
{
    print [[$* cString]
}

function prstar
{
    print -r *($*)
}

function pcounts # print retain count and number of autoreleases of
$1
{
    print [[$* retainCount]
    print [NSAutoreleasePool _numberOfObjectsIdenticalTo: $*]
}

# print string of an NSText object
```



```
dalias ptext print [[!1 text] cString] // sets dbx alias ptext to
                                        print string of an NSText object

# print string of an NSCStringEncoding object
dalias pcs  print [[!1 cStringEncodingInternalState]->_string cString]
                                        // sets dbx alias pcs to print string of
                                        an NSCStringEncoding object

dalias pns pnsstring // sets dbx alias pns for pnsstring command
dalias pd  pdesc     // sets dbx alias pd for pdesc command
dalias prs prstar    // sets dbx alias prs for prstar command

alias typeof="print -l ((NSObject *)!:*)->isa->name" // sets dbx
alias typeof for printing type of
current object

dalias currwin "print -l [(NSView *)[NSView focusView] window]"
// sets dbx alias currwin for printing
current window

dalias flushcurrwin "print -l [(NSWindow *)[NSView *)[NSView
focusView] window]) _forceFlushWindowToScreen]" // sets dbx alias
flushcurrwin for synchronous flushing
of current window's off-screen buffer
to screen

button expand whatis // adds whatis button command;if selected
characters begin with alphanumeric
character, $, or _, then expands
selection and uses as target
button expand prstar // adds prstar button command;if selected
characters begin with alphanumeric
character, $, or _, then expands
selection and uses as target
button expand pselfvar // adds pselfvar button command;if
selected characters begin with
alphanumeric character, $, or _, then
expands selection and uses as target
button expand pnsstring // adds pnsstring button command;if
selected characters begin with
alphanumeric character, $, or _, then
```

```

                                expands selection and uses as target
button expand pcounts           // add pcounts button command;if selected
                                characters begin with alphanumeric
                                character, $, or _, then expands
                                selection and uses as target
button expand pdesc             // add pcounts button command;if selected
                                characters begin with alphanumeric
                                character, $, or _, then expands
                                selection and uses as target
button ignore defbrks          // adds defbrks button command; ignores
                                current mouse selection for command

#####
#                               General purpose settings                               #
#####

toolenv cmdlines 20            // sets the number of lines in the
                                command subwindow to 20
dbxenv step_events on          // sets the dbx customization variable
                                step_events to on to allow breakpoints
                                while stepping or "nexting"
dbxenv suppress_startup_message 4.0 // sets the dbx customization
                                variable suppress_startup_message to
set -o ignoresuspend           # uncomment to cause dbx to ignore ^Z
set -o emacs                   # uncomment to enable emacs-style command
                                editing
#set -o vi                     # or uncomment this line for vi-style editing

function attach # attach to a running process
{
    typeset PIG="$(/bin/ps -ef | /bin/egrep ${1} | /bin/egrep -v
egrep | /bin/head -1 | /bin/awk '{ print $8 " " " $2 }')
    debug $PIG
}

function collOn # enable collector modes
{
    collector work_set mode on
}

```

```
        collector profile mode stack
    }

function ff
{
    where -f $(frame) 1
    list
}

function penviro  # dump the environment variables of the target
process
{
    [ -z "$1" ] || { echo "$0: unexpected argument" >&2 && return; }
    typeset -i i=0
    typeset env="((char **)$[(char**)environ]"
    while :
    do
        x=${env}[i]
        echo "$i: " "${x#0x*\ }"
        case "$x" in
            *\ (nil\ )*)    break;;
        esac
        ((i += 1))
    done
}

PS1="$SMSO(dbx !)$RMSO " # reverse-video prompt with history number

function _cb_prompt
{
    if $mtfeatures
    then    # set prompt for MT debugging
        PS1='${SMSO}${thread} ${lwp}:!${RMSO} '
    fi
}
```

```

else      # set prompt for non-thread debugging
    PS1='${SMSO}dbx:!${RMSO} '
fi
}

function hex      # print arg in hex
{
    : ${1?"usage: $0 <expr> # print <expr> in hex"}
    typeset -i16 x
    ((x = ${int}$*))
    echo - $* = $x
}
typeset -q hex

function hexdump      # dump $2 (default: sizeof $1) bytes in hex
{
    : ${1?"usage: $0 <exp> [<size>] # dump <size> bytes in hex"}
    typeset -i16 p="${(void *)&$1}" # address of $1
    typeset -i s="${2:-${sizeof ($1)}}" >/dev/null 2>&1 # number of
bytes
    builtin examine $p/${(s:-4)+3}/4X
}
typeset -q hexdump

function pg # print process status by name
{
    /bin/ps -ef | /bin/egrep ${1} | /bin/egrep -v egrep
}

regs() # print register contents
{
    case $# in
    0) x &&$g0/32X; x &&$y/X; x &&$psr/X; x &&$pc/X; x &&$npc/X ;;
    *) for i

```

```
do reg=\$$i; x &$reg/X
done ;;
esac
}

dalias p print // sets dbx alias p for print command
dalias w where // sets dbx alias w for where command
dalias br where // sets dbx alias br for where command
dalias ww where -q // sets dbx alias ww for where -q (quick
traceback) command

dalias fr frame // sets dbx alias fr for frame command
dalias b stop in // sets dbx alias b for stop in command
dalias ba stop at // sets dbx alias ba for stop at command
dalias si stop in // sets dbx alias si for stop in command
dalias sa stop at // sets dbx alias sa for stop at command
dalias sic stop inclass // sets dbx alias sic for stop inclass
command
dalias sif stop infunction // sets dbx alias sif for stop
infunction command
dalias sim stop inmember // sets dbx alias sim for stop inmember
command
dalias sm stop modify // sets dbx alias sm for stop modify
command
dalias sr stop returns // sets dbx alias sr for stop returns
command
kalias cc="clear;cont" // sets Korn alias cc for clear command
followed by cont command
dalias cl clear // sets dbx alias cl for clear command
dalias ib status // sets dbx alias ib for status command
dalias st status // sets dbx alias st for status command
dalias d delete // sets dbx alias d for delete command
dalias r run // sets dbx alias r for run command
dalias c cont // sets dbx alias c for cont command
dalias s step // sets dbx alias s for step command
dalias su step up // sets dbx alias su for step up command
```

```
dalias n next          // sets dbx alias n for next command
dalias di handler -disable // sets dbx alias di for handler
                        -disable command
dalias en handler -enable // sets dbx alias en for handler -enable
                        command
dalias N nexti        // sets dbx alias N for nexti command
dalias S stepi        // sets dbx alias S for stepi command
dalias q quit         // sets dbx alias q for quit command
dalias tiny toolenv srclines 16; toolenv cmdlines 8 // sets dbx alias
                        tiny to 16 lines in the source
                        subwindow and 8 lines in the command
                        subwindow
dalias mid toolenv srclines 25; toolenv cmdlines 25 // sets dbx alias
                        mid to 25 lines in the source
                        subwindow and 25 lines in the command
                        subwindow
dalias big toolenv srclines 33; toolenv cmdlines 14 // sets dbx alias
                        big to 33 lines in the source
                        subwindow and 14 lines in the command
                        subwindow

dalias und_all undisplay
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20 // sets dbx
                        alias und_all to undo display commands
                        1 through 20
dalias insense dbxenv case insensitive // sets dbx alias insense to
                        make case insignificant in variable
                        and function names
dalias sense dbxenv case sensitive // sets dbx alias sense to make
                        case significant in variable and
                        function names

button lineno cont at // adds button command cont at; uses
                        line number associated with current
                        selection as target of command
button ignore step up // adds button command step up; ignores
                        current mouse selection for command
button ignore tiny // adds button command tiny; ignores
                        current mouse selection for command
button ignore mid // adds button command mid; ignores
                        current mouse selection for command
```

```
button ignore big           // adds button command big; ignores
                             current mouse selection for command
button ignore quit         // adds button command quit; ignores
                             current mouse selection for command
```

Helpful User Default Variables to Set with `dwrite`

The following user default variables, which you can set with the `dwrite` command, may be useful in debugging your application:

```
NSEnableAutoreleasePool
NSEnableDoubleReleaseCheck
NSHideOnDeactivateEnabled
NSPauseAtStartup
NSSetPoolThreshold
NSShowAllWindows
NSShowAllViews
NSShowAllWindows
NSShowDrawTimes
NSShowEvents
NSShowPS
NSShowWindowInfo
NSShowXEvents
NSTrapIllegalFloatingPointOps
```

Tracing Objective C Objects

You can monitor Objective C messages being sent by objects in your application by calling the function `objc_messageSendDebug`. This function allows you to filter on different message attributes, and also stop at breakpoints when a certain filter matches the current message.

This facility is particularly useful for finding memory allocation errors and performance problems.

You can enable `messageSendDebug` in three ways:

- By using the `NSUserDefaults` system and `dwrite` commands (see “Invoking `messageSendDebug` Using `dwrite` Commands”)
- From a Debugger command line or within a program, by calling the functions found in `/usr/openstep/include/objc/objc-debug.h` (see “Invoking `messageSendDebug` from a Program or the Debugger” on page A-15)
- From a graphical tool which supports remote `messageSend` debugging, such as the `ObjectDebug` tool included in `OpenStep`

Invoking `messageSendDebug` Using `dwrite` Commands

You can automatically invoke `messageSendDebug` at execution time by using one of the following `dwrite` commands.

This command turns on `messageSendDebug`, but no messages are sent until a filter is set:

```
dwrite AppName NSEnableMessageSendDebug YES
```

This command suspends the display of messages, even if filters are set:

```
dwrite AppName NSEnableMessageSendDebug NO
```

This command turns on `messageSendDebug`, and adds a generic filter to show all messages:

```
dwrite AppName NSEnableMessageSendDebug ALL
```

This command displays an explanation of how to use this facility:

```
dwrite AppName NSEnableMessageSendDebug HELP
```

Adding Individual Message Filters

You can add individual message filters with the following commands:

```
dwrite AppName NSMessageSendDebugFilter "ClassName | *,  
[+ | -]selectorName | [+ | -]*, receiverID[hex or dec] | *, YES | NO  
dwrite AppName NSMessageSendDebugFilter "GENERIC_FILTER"  
dwrite AppName NSMessageSendDebugFilter1 "(AnotherSelectorName,  
...)"  
dwrite AppName NSMessageSendDebugFilterN "(SelectorNameN, ...)"
```


YES or NO applies to whether or not to call `objc_messageMatchedFilter()` when a filter matches. Enter YES if you want your program to hit a breakpoint when any filter matches (see “Setting a Breakpoint on a Filter Match” on page A-17).

GENERIC_FILTER shows all messages.

If `ClassName` in the filter is *, any class counts as a match.

If `selectorName` in the filter is *, any selector counts as a match.

If `selectorName` in the filter is preceded by a "+" or "-", only class methods, or instance methods (respectively) are considered matches.

If `receiverID` in the filter is *, any receiver counts as a match.

If `ClassName`, `selectorName`, and `receiverID` are all *, all messages are considered matches.

Controlling Call Level Indentation

By default, the call level (nested level) of each method is shown in the matched filter output by indenting the line. At times this may be undesirable. To disable or enable this feature, use one of the following commands to control (typically turn off) call level indentation in all applications.

```
dwrite AppName NSEnableFilterCallLevelIndentation YES | NO
dwrite NSGlobalDomain NSEnableFilterCallLevelIndentation YES | NO
```

This setting effects console output only, and has no effect on external debug-monitoring applications like ObjectDebug.

Invoking `messageSendDebug` from a Program or the Debugger

For detailed information, see the file `/usr/openstep/include/objc/objc-debug.h`. Enabling `messageSendDebug` adds to, and does not preclude, filtering options you have set using `dwrite`.

Enabling `messageSendDebug`

To enable `messageSendDebug`, call one of the following methods:

```
objc_enableMessageSendDebug(EnableDebug)
```

```
objc_enableMessageSendDebug(EnableDebugShowAllMessages)
objc_enableMessageSendDebug(EnableDebugSilently)
objc_enableMessageSendDebug(DisableDebug)
objc_enableMessageSendDebug(DisableDebugSilently)
```

The following methods are equivalent to the above methods:

```
objc_enableMessageSendDebug(1)
objc_enableMessageSendDebug(2)
objc_enableMessageSendDebug(3)
objc_enableMessageSendDebug(0)
objc_enableMessageSendDebug(-1)
```

Note – You may want to disable this mechanism before making method calls in your debugger, as those method calls will get traced as well!

Adding Filters

To add filters, you can call one of the following methods:

```
objc_addFilterFromString(const char *filterString)
objc_addFilterForClass(const char *className)
objc_addFilterForSelector(const char *selectorName)
objc_addFilterForReceiver(id receiver)
```

`objc_addFilterFromString` has the same syntax as the `NSMessageSendDebugFilter` `dwrite` command, with the addition of a `FilterID` field as the first value. This field lets you uniquely identify the filter.

The filter string format looks like this:

```
objc_addFilterFromString("FilterID, ClassName | *,
[+ | -]selectorName | [+ | -]*, receiverID[hex or dec] | *, YES | NO
```

or this:

```
objc_addFilterFromString("GENERIC_FILTER")
```

`YES` or `NO` applies to whether or not to call `objc_messageMatchedFilter()` when a filter matches. Enter `YES` if you want your program to hit a breakpoint when any filter matches (see “Setting a Breakpoint on a Filter Match” on page A-17).

GENERIC_FILTER shows all messages.

If `ClassName` in the filter is `*`, any class counts as a match.

If `selectorName` in the filter is `*`, any selector counts as a match.

If `selectorName` in the filter is preceded by a "+" or "-", only class methods, or instance methods (respectively) are considered matches.

If `receiverID` in the filter is `*`, any receiver counts as a match.

If `ClassName`, `selectorName`, and `receiverID` are all `*`, all messages are considered matches.

Controlling Call Level Indentation

By default, the call level (nested level) of each method is shown in the matched filter output by indenting the line. At times this may be undesirable. To disable or enable this feature, call the following method:

```
objc_enableFilterCallLevelIndentation(0 | 1)
```

This setting effects console output only, and has no effect on external debug-monitoring applications. It is unnecessary if the feature has already been enabled or disabled with `dwrite`.

Removing Filters

To remove all filters, call the following method:

```
objc_removeAllFilters()
```

Disabling Filters

To temporarily disable all filters, call the following method:

```
objc_enableMessageSendDebug(DisableDebug[0])
```

Setting a Breakpoint on a Filter Match

If you want your program to hit a breakpoint when any filter matches, call the following method:

```
objc_callMessageMatchedFilter(0 | 1)
```

Note - `objc_callMessageMatchedFilter` sets this flag for all existing filters. To set the flag for individual filters, use the `objc_addFilterFromString` method to create your filter, or call `objc_callMessageMatchedFilter` after setting some filters, and then set the rest of your filters.

Once you continue program execution, a string is printed indicating that the current message or receiver matched one of the set filters.

After this string is printed, the function `objc_messageMatchedFilter()` is called by the Objective C runtime system.

You can put a breakpoint at `objc_messageMatchedFilter()` to get a backtrace.

Examples

Example 1:

To see all the messages sent to the `NSAutoreleasePool` class, either enter the following `dbx` commands in the Debugger Command Pane:

```
call objc_enableMessageSendDebug(1)
call objc_addFilterForClass("NSAutoreleasePool")
```

or use the following `dwrite` commands at execution time:

```
dwrite AppName NSEnableMessageSendDebug YES
dwrite AppName NSMessageSendDebugFilter "NSAutoreleasePool,*,*,NO"
```

Example 2:

To see all the `addObject:` messages sent to the `NSAutoreleasePool` class, and have Objective C call `objc_messageMatchedFilter()` when that message is sent (so you can hit a breakpoint there), either enter the following `dbx` commands in the Debugger Command Pane:

```
call objc_enableMessageSendDebug(1)
call
objc_addFilterFromString("1,NSAutoreleasePool,addObject:*,*,YES")
```

or use the following `dwrite` commands at execution time

```
dwrite AppName NSEnableMessageSendDebug YES
dwrite AppName NSMessageSendDebugFilter1
"NSAutoreleasePool,addObject:,* ,YES"
```

Notice the lack of the first parameter, the `filterID`, which is automatically generated in this case by the number that is appended to the `dwrite key + 1000` (for example. "1001").

Example 3:

To see all the class method calls (as opposed to instance method calls) sent to any object, and not show call level indentation, either enter the following `dbx` commands in the Debugger Command Pane:

```
call objc_enableMessageSendDebug(1)
call objc_addFilterForSelector("+*")
call objc_enableFilterCallLevelIndentation(0)
```

or use the following `dwrite` commands at execution time:

```
dwrite AppName NSEnableMessageSendDebug YES
dwrite AppName NSMessageSendDebugFilter "* ,+* ,* ,NO"
dwrite AppName NSEnableFilterCallLevelIndentation NO
```

Implementing Your Own Filtering Mechanism

If you want to implement your own filtering mechanism, you can call the following function:

```
objc_setMessageSendFilterFunction(void
(*customFilterFunction)(Class receiverClass,id receiver, SEL
selector,void *callLevel, void *threadID))
```

This function takes a pointer to a `filterFunction`. After calling this function, every message (`objc_msgSend`) that is sent will go through your own filter function. You can then implement your own filtering system.

You can also call the following function from your filter function, in case you want to do the normal filtering stuff but tweak a few things first.:

```
objc_defaultMessageSendFilterFunction()
```

You can get a linked list of all the currently set filters by calling the following:

```
objc_filterList(void)
```

Debugging Applications Using Optimized Libraries

If you compile an application with `-g` but use libraries not compiled with `-g`, the Debugger does not know the prototypes of methods defined in the libraries. This means it does not know the types of the returned values, nor of the parameters. It assumes the types are `int`.

For example, assume that `set1` is an `NSSet`. You could use casts to tell the Debugger the return types of methods:

```
dbx:3 whatis set1
@interface NSSet *set1;
dbx:4 p [set1 description]
[set1 description] = -283014864
dbx:5 p (NSString *)[set1 description]
(@interface NSString *) [set1 description] = 0xef218bd0
dbx:6 p [(NSString *)[set1 description] cString]
[(@interface NSString *) [set1 description] cString] = -281204711
dbx:7 p (char *)[(NSString *)[set1 description] cString]
(char *) [(@interface NSString *) [set1 description] cString] =
0xef3d2841 "NSConcreteSet(a, b)"
```

In order to obviate the need for these casts, Project Builder includes a special module named `dbxInfo.o`. This module contains debugging information for all the methods defined in the Application Kit and Foundation Kit libraries. When you do a debug build using the Project Builder Makefiles, this module is automatically linked into your application. In order to make this information available to the Debugger after it has started, Project Builder issues the following command to the Debugger as it is starting up:

```
module dbxInfo.o
```

This command causes the Debugger to read in the debugging information contained in `dbxInfo.o` so it is available when the Debugger has to determine the return types and parameter types of methods invocations.

Interface Builder Application Programming Interface



This appendix describes the application programming interface (API) that lets you build custom palettes, inspectors, and editors for Interface Builder.

Interface Builder gives you direct access to the majority of the objects defined in OpenStep. For example, you can easily add an `NSText` object to your application—an object that represents years of programming and testing effort—by dragging the object from Interface Builder’s Palette window into your application’s window. By creating a custom palette containing objects of your own design, you and other developers can manipulate these objects as easily as you do the ones in Interface Builder’s standard palettes.

Using the facilities described in this appendix, you can easily create a palette that contains one or more objects of your own design. These objects can be of various types:

Type	Instantiation
NSView objects	Can be dragged into one of the application’s standard windows.
NSMenuItem objects	Can be dragged into one of the application’s menus.
NSWindow objects	Can be dragged into the workspace.
Other non-NSView objects	Can be dragged into Interface Builder’s File window.

The API described here also lets you provide inspectors for any custom object. There are four kinds of inspectors: Attributes, Connections, Size, and Help. The most common inspector to implement is the Attributes inspector, which lets the user set the custom object’s unique features. For example, if you define a custom

button object that sends a message repeatedly when it is pressed, the Attributes inspector could let the user set the repeat rate. Objects with special connection requirements can provide their own Connection inspectors. The Size and Help inspectors are rarely overridden since they are appropriate for most types of objects.

If you need to provide the user with a more sophisticated system for interacting with your custom objects, you can implement an *editor* using the API described in this appendix. Whereas an inspector borrows Interface Builder's Inspector panel for its display, an editor provides its own window. The size of this window is not constrained as is the inspector window. Since each object can have its own editor, there can be multiple editor windows on the screen at once, making "copy and paste" and "drag and drop" interactions possible between editor windows. If the edited object contains other objects, the editor can open subeditors to let the user interact with the contained objects.

To provide a better context for the discussion of the programming interface that makes custom palettes, inspectors, and editors possible, the next section gives a broad overview of Interface Builder's design.

Interface Builder's Design

You use Interface Builder to assemble and interconnect your application's objects. You start the process by creating a new document (or, more likely, by modifying the default document provided by Project Builder). When you save the document, it is represented by a file package having a name ending in `.nib`.

An Interface Builder document contains the following:

- An object hierarchy
- References to custom classes
- Connection information
- Resources such as sounds and images

Within Interface Builder, these components are managed by a *document object*. This object is of a private class, but can be queried and updated through the methods declared in the `IBDocuments` protocol.

The Object Hierarchy

A document object stores and maintains an object hierarchy. At the top of the hierarchy is the File's owner object—the object that is represented in the top-left portion of the File window. This is actually a proxy object, since the actual object that owns the interface exists outside of the nib file. When a user adds an object to the interface project, it becomes part of the document by being attached to some other object—the *parent* object—in the object hierarchy. (In this hierarchy, a parent object may have many children, but each child can have only one parent object.) An object must be part of this hierarchy for it to be archived in the nib file.

Interface Builder declares and implements several methods as a category of `NSObject` (see “NSObject Additions” on page C-11) so that it can query any object in the hierarchy for crucial information. For example, each object can identify its various inspectors since it inherits the following methods:

- `connectInspectorClassName`
- `sizeInspectorClassName`
- `helpInspectorClassName`
- `inspectorClassName`

When you define a class for a custom palette object, you can override any of these methods to provide your own inspectors.

Class References

Often, the object you want instantiated when your application runs is not available to Interface Builder either from its own library of objects or from any palette that has been dynamically loaded. For these cases, Interface Builder provides a proxy object such as the `NSCustomView` object in the Basic Views palette. When you drag an `NSCustomView` into your application, you are in fact adding this proxy object to the document's object hierarchy. When the resulting nib file is loaded within a running application, the proxy object is unarchived and queried to determine the identity of the class that the proxy represents. Then, an instance of this custom class is created (through the facilities of the `alloc` and `init` messages), and the proxy is freed.

Note that this distinction between objects that are unarchived and objects that are represented by proxies has important consequences. An object that is unarchived can receive `awakeAfterUsingCoder:` message, but does not receive an `init` message. On the other hand, an object that is represented by a proxy object in the nib file receives only an `init` message—it does not receive an `awakeAfterUsingCoder:` message.

Connection Information

An Interface Builder document also contains information about how objects within the object hierarchy are interconnected. This connection information is embodied in objects that conform to the `IBConnectors` protocol. Each connector object stores information about a connection between one source object and one destination object. Interface Builder's Connections inspector is the interface to a document's connector objects. Each time you connect a source object with a destination object, you are creating another connection object.

When you save the document, connector objects are archived in the nib file along with the objects they interconnect. When an application loads the nib file, the objects from the object hierarchy are unarchived, proxy objects are replaced with the appropriate instances, and connection objects are unarchived. The application then sends each connection object an `establishConnection` message, giving it an opportunity to connect its source and destination as it deems appropriate. The standard connection object that Interface Builder provides (again, of an unspecified class) stores the identity of the source object's outlet variable and the destination object's action method, if any. So, when such a connector object receives an `establishConnection` message, it sets the source object's outlet to the destination object and—if the source object's outlet is named "target"—it sets the source's action to the destination's action method.

In most cases, Interface Builder's standard connection objects will be sufficient for your needs. However, you can create a Connection inspector and connection objects of your own, and through the methods declared in the `IBDocuments` protocol, you can have these connection objects archived in the nib file.

Interface Builder's Programming Interface

The API that Interface Builder defines is organized as two class definitions, several protocols, and several methods that are added, through the use of categories, to the definitions of the `NSObject` and `NSView` classes. The functions of these components are summarized in the following sections.

Classes

Interface Builder uses the class definitions for `IBPalette` and `IBInspector` as links to your custom palettes and inspectors. It is through the methods defined in these classes that Interface Builder locates and loads the user-interface objects that appear in the custom palette and in the inspector for a custom object. These classes are described in detail in Appendix C, "Interface Builder API Classes."

Class	Description
<code>IBPalette</code>	This class is provided as the owner of palette's interface. You must create a subclass of <code>IBPalette</code> to associate the images in the Palette window with the real objects you intend to have instantiated.
<code>IBInspector</code>	This is the abstract superclass for inspectors. Your inspector provides Interface Builder with the controls to be loaded into the Inspector panel when the user attempts to inspect the custom object. The inspector also interprets the user's actions on these controls as commands to modify the custom object's state.

Protocols

The protocols listed below define the ways your dynamically loaded palettes and inspectors can communicate with Interface Builder (the `IB` and `IBDocuments` protocols) and the ways Interface Builder can communicate with objects in your module (the remaining protocols). These protocols are described in detail in Appendix D, “Interface Builder API Protocols.”

Protocol	Description
<code>IB</code>	This protocol gives you access to global information: the object that represents the active document, whether Interface Builder is in test mode, the source and destination objects of a connection, and so on.
<code>IBDocuments</code>	This protocol defines the programming interface to a document object in Interface Builder. Through this interface, you can add and remove objects from the document’s object hierarchy, add or remove a connector object, and set the active editor.
<code>IBEditors</code>	This protocol declares the methods through which Interface Builder can interact with an editor object. Interface Builder invokes these methods to make the editor’s selected object visible; to copy, paste, or delete the selection; and to open and close subeditors, among other things.
<code>IBSelectionOwners</code>	Editor objects conform to this protocol, which declares methods for counting the number of objects in the selection and for filling an <code>NSArray</code> object with the objects in the selection.
<code>IBConnectors</code>	This protocol declares the methods that connector objects must implement. These include methods for identifying the source and destination of a connection and for establishing the connection between these objects.

Other Programming Interfaces

Through the use of categories, Interface Builder adds methods to the `NSObject` and `NSView` classes. These methods are described in “`NSObject` Additions” on page C-11 and “`NSView` Additions” on page C-15.

Class	Description
NSObject Additions	Interface Builder uses these methods to discover the various inspectors for the selected object. Default inspectors and editors are provided for all objects.
NSView Additions	These methods let custom <code>NSView</code> objects control how they are resized and redrawn.
NSCell Additions	These methods let custom <code>NSCell</code> objects control how they are resized and redrawn.

Interface Builder API Classes



IBInspector

Characteristic	Description
Inherits From:	NSObject
Declared In:	include/InterfaceBuilder/IBInspector.h

Class Description

The `IBInspector` class defines the interface between an inspector for a loadable module and the Interface Builder application. When you build a new inspector for Interface Builder, you create a subclass of `IBInspector`.

The inspector you define must load its interface (that is, the nib file containing the interface), and it must override the `ok:`, `revert:`, and `wantsButtons` methods. The nib file is generally loaded as part of the inspector's `init` method. The `wantsButtons` method controls whether the inspector displays OK and Revert button. (As with Interface Builder's standard inspectors, most custom inspectors do not need these buttons—instead, the user's actions in the Inspector panel are registered immediately by the inspected object.) The `ok:` and `revert:` methods control the synchronization of the Inspector panel's state with that of the inspected object. Interface Builder sends the inspector a

`revert:` message to make the inspector reflect the current state of the inspected object. The `ok:` message should cause the inspector to set the state of the inspected object to that displayed in the Inspector panel.

An inspector should send itself a `touch:` message when the user begins modifying the data it displays. This message displays a broken “X” in the panel’s close box and enables the inspector’s OK and Revert buttons, if present. (See “`textDidBeginEditing:`” on page C-5 for alternate way to achieve this result.)

Instance Variables

```
id object;  
NSWindow* window;  
id manager;  
NSButton* okButton;  
NSButton* revertButton;
```

Variable	Description
<code>object</code>	The object that is being inspected
<code>window</code>	The Panel that contains the inspector’s user interface
<code>manager</code>	private
<code>okButton</code>	The Inspector panel’s OK button, if present
<code>revertButton</code>	The Inspector panel’s Revert button, if present

Method Types

Activity	Class Methods
Accessing objects	<ul style="list-style-type: none">- object- okButton- revertButton- textDidBeginEditing- wantsButtons- window
Managing changes	<ul style="list-style-type: none">- ok:- revert- textDidBeginEditing- touch

Instance Methods

`object`

- (id)object

Returns the object that is being inspected.

`ok:`

- (void)ok:(id)sender

Implement in your subclass of `IBInspector` to commit the changes that the user makes in the Inspector panel. The OK button in the Inspector panel—if present—sends an `ok:` message when the user clicks on it.

Your implementation of this method must send the same message to `super` as follows:

```
(void)ok:(id)sender
{
    /* your code to commit changes */
    [super ok:sender];
    return self;
}
```

The message to `super` replaces the broken “X” in the panel’s close box with the standard “X”, indicating that the changes have been committed. See also `- revert:`, `- touch:`.

`okButton:`

```
- (NSButton *)okButton
```

Returns the Inspector’s OK button object. This can be useful if you want to alter its title, for example. See also `- revertButton`.

`revert:`

```
- (void)revert:(id)sender
```

Implement in your subclass of `IBInspector` to synchronize the inspector’s display with the state of the object being inspected. Interface Builder sends this message to the inspector object whenever the inspector’s display might need to be updated, for example, when the user opens the Inspector panel and the selected object in Interface Builder is of the type associated with this inspector. The Revert button in the Inspector panel—if present—also sends a `revert:` message when the user clicks on it.

Your subclass must implement this method, and it must send the same message to `super` as part of its implementation, as follows:

```
revert:sender
{
    /* your code to inspect selected object */
    [super revert:sender];
    return self;
}
```

This message to `super` replaces the broken “X” in the panel’s close box with the standard “X”, indicating that the changes have been discarded. See also `- ok:`, `- touch`.

revertButton:

- (NSButton *)revertButton

Returns the Inspector's Revert button object. This can be useful if you want to alter its title, for example. See also - okButton.

textDidBeginEditing:

- textDidBeginEditing:sender

Sends the IBInspector a touch: message on behalf of some NSText object in the Inspector panel.

By making your inspector object the delegate of any NSText object in the Inspector panel, the panel will be updated appropriately as the user alters the panel's contents. See also - touch:.

touch:

- (void)touch:(id)sender

Changes the image in the Inspector panel's close box to a broken "X" to indicate that the contents have been edited. Also, enables the buttons that allow the user to commit or abandon changes. See also - textDidBeginEditing:.

wantsButtons

- (BOOL)wantsButtons

Returns a boolean value indicating whether the inspector object requires Interface Builder to display the OK and Revert buttons in the Inspector panel.

window

- (NSWindow *)window

Returns the NSWindow object that contains the user interface for the inspector.

IBPalette

Characteristic	Description
Inherits From:	NSObject
Declared In:	include/InterfaceBuilder/IBPalette.h

Class Description

The `IBPalette` class defines Interface Builder's link to a dynamically loaded palette. Interface Builder uses the facilities of this class to load a custom palette's interface and executable code.

Each loadable palette must contain a subclass of `IBPalette`, and this class must be identified in the palette's `palette.table` file. Interface Builder creates an instance of this subclass when it loads the palette. In its `init` method, the subclass must load the nib file containing the classes of objects on the palette. Interface Builder then sends the palette object an `originalWindow` message to access the window that contains the views to be displayed in the Palette window.

If a palette contains non-`NSView` objects (`NSMenuCells`, `NSWindows`, or objects that will be deposited in the nib file window), the subclass must implement the `finishInstantiate` method to associate each `NSView` object that is displayed in the Palette window with the non-`NSView` object that should be created when the user instantiates the object by dragging it from the palette.

For example, consider a custom palette that provides an `AddressBook` object that manages people's names and addresses. This object, a subclass of `NSObject`, is to be dragged into the nib file window. Further, imagine that the subclass of `IBPalette` for this custom palette, `AddressBookPalette`, has two outlets: `addressBookObject` and `addressBookView`. When the palette's nib file was created, the outlets were connected to the `AddressBook` object and to an `NSView` object that will represent it in the Palette window. Within the `AddressBookPalette` class implementation file, the `finishInstantiate` method would look like this:

```
- (void)finishInstantiate
{
    [self associateObject:addressBookObject
```

```

        type:IBObjectPboardType with:addressBookView];
    return self;
}

```

Notice that the subclass establishes an association by sending itself an `associateObject:ofType:withView:` message. `IBPalette` implements this method. The second argument is a type string defined in `IBPalette.h` that controls where the palette image may be deposited:

Type	Usage
<code>IBObjectPboardType</code>	For objects that the user must deposit in the File window
<code>IBMenuCellPboardType</code>	For <code>NSMenuCells</code> without submenus; must be deposited in a menu
<code>IBMenuPboardType</code>	For <code>NSMenuCells</code> that have submenus; must be deposited in a menu
<code>IBWindowPboardType</code>	For <code>NSWindows</code> and <code>NSPanels</code> ; must be deposited in the workspace

Instance Variables

```

id <IBDocuments> _paletteDocument;
NSWindow *_originalWindow;
IBPaletteView *_paletteView;
NSView *_draggedView;
id _paletteBundle;

```

Variable	Description
<code>paletteDocument</code>	An object conforming to the <code>IBDocuments</code> protocol that represents the dynamically loaded palette
<code>originalWindow</code>	The window containing the interface objects that will be loaded into Interface Builder's Palettes window
<code>paletteView</code>	private
<code>draggedView</code>	private
<code>paletteBundle</code>	private

Method Types

Activity	Class Methods
Associating NSViews and NSObjects	– <code>associateObject:ofType:withView:</code>
Initializing the palette	– <code>finishInstantiate</code>
Accessing related objects	– <code>paletteDocument</code> – <code>originalWindow</code> – <code>imageName:</code>

Instance Methods

`associateObject:ofType:withView:`

```
– (void)associateObject:(id)object ofType:(NSString *)type
withView:(NSView *)view
```

Establishes an association between an `NSView` in a palette (`view`) and the object that should be instantiated when the user drags the `NSView` from the palette (`object`). The `type` argument controls where the palette object may be deposited. (See “Class Description” on page C-6 for more information.)

If your custom palette provides non-`NSView` objects, override `IBPalette`’s `finishInstantiate` method with an implementation that sends `associateObject:ofType:withView:` messages to associate each `NSView` object in the palette with the non-`NSView` object that it represents.

`imageName:`

```
– (NSImage *)imageName:(NSString *)name
```

Returns the `NSImage` instance associated with `name`. If no such image can be found, this method returns `nil`.

Use this method to refer to images in your custom palette. This method first tries to find the image by invoking `NSImage`’s version of `findImageNamed:`. If that is unsuccessful, it uses the facilities of the `NSBundle` class to check the `.palette` directory for this resource. See `getPath:forResource:ofType:` for a description of `NSBundle`’s search path. See also
– `pathForResource:ofType:` (`NSBundle` common class).

`finishInstantiate`

- (void)finishInstantiate

Implement to complete the initialization of your `IBPalette` object. Interface Builder sends a `finishInstantiate` message to the `IBPalette` object after it has been unarchived from the palette file. A typical use of this method is to associate an `NSView` object within the custom palette with a non-`NSView` object that is meant to represent it in the Palette window. See “Class Description” on page C-6 for more information. See also - `associateObject:type:with:.`

`originalWindow`

- (NSWindow *)originalWindow

Returns the `NSWindow` that contains the `NSView` objects to be loaded into Interface Builder’s Palette window. When it loads a custom palette, Interface Builder sends the `IBPalette` subclass an `originalWindow` message. In your custom palette, you must connect the `originalWindow` outlet of your subclass of `IBPalette` to the `NSWindow` that contains the `NSViews` that represent your palette objects.

`paletteDocument`

- (id<IBDocuments>)paletteDocument

Returns an object that represents the dynamically loaded palette. This object is of unspecified class; however, it conforms to the `IBDocuments` protocol.

NSApplication Additions

Characteristic	Description
Category Of:	<code>NSApplication</code>
Declared In:	<code>include/InterfaceBuilder/IBConnectors.h</code>

Category Description

Interface Builder adds these methods to the definition of the `NSApplication` class for managing connections. They may be useful when implementing your own Connections inspector.

Instance Methods

`connectDestination`

- (id)connectDestination

Returns the object that is the destination of the connection; that is, the object to which the user has dragged a connection line. See also - `connectSource`.

`connectSource`

- (id)connectSource

Returns the object that is the source of the connection; that is, the object from which the user has dragged a connection line. See also - `connectDestination`.

`displayConnectionBetween:and:`

- (void)displayConnectionBetween:(id)source and:(id)destination

Causes Interface Builder to draw connection lines between source and destination. For example, when the user clicks on an entry in the Connections list in the Connections inspector, Interface Builder uses this method to display the corresponding connection.

The act of displaying a connection between these two objects does not require that a connection really exist, and does not create a connection. It is the Connection inspector's responsibility to establish the programmatic connection. This method simply draws lines between two objects and attempts to make both objects visible. See also - `stopConnecting`.

`isConnecting`

- (BOOL)isConnecting

Returns YES if connection lines are being displayed in Interface Builder. You can use this information to control how your object is drawn during the connection process. For example, when you drag a connection line from a button, the button's black border and text are redrawn in gray. See also - `stopConnecting`.

stopConnecting

- (void)stopConnecting;

Causes Interface Builder to remove any connection lines from the screen. Interface Builder uses this method to remove connection lines when the user drags a window. See also - isConnecting.

NSObject Additions

Characteristic	Description
Category Of:	NSObject
Declared In:	include/InterfaceBuilder/IBDocuments.h include/InterfaceBuilder/IBEditors.h include/InterfaceBuilder/IBInspector.h include/InterfaceBuilder/IBObjectAdditions.h

Category Description

Interface Builder adds these methods to the definition of the `NSObject` class so that any palette object can be queried for its inspectors, for its editor, and for an image to represent the object when it is instantiated in the File window.

The methods described in “Instance Methods” on page C-12 return the class name for the object that will own the Inspector panel’s display. Interface Builder caches this information so that when the user attempts to inspect an object, Interface Builder knows what type of inspector to instantiate (if it has not yet been instantiated). The inspector object provides the interface that Interface Builder displays in the Inspector panel.

Interface Builder supplies default implementations of these methods; you only override them if your custom palette object requires it. For example, if you create an `NSTextField` palette object that validates its input, you would probably provide an Attributes inspector that lets the user specify the acceptable input values. Thus, you would override the `inspectorClassName` method to return the class name for the Attributes inspector object. However, you would probably not have to override the other inspector methods since the standard inspectors would be satisfactory.

By default, these methods return the empty string “”, except for `imageforViewer`, which returns `nil`.

Override the `editorClassName` method to return the class name of the editor to use for this object. The editor is invoked when the user double-clicks on the object. `NSView` objects inherit Interface Builder's standard `NSView` editor.

You only need to override the `imageForViewer` method if you want a special image to represent your custom palette object when it is dragged into the nib file window. If you do not supply such an image, Interface Builder uses the standard cube image.

Instance Methods

`awakeFromDocument:`

- (void)awakeFromDocument:(id<IBDocuments>)document

Allows special behavior after the nib document has been loaded.

`canSubstituteFor Class:`

+ (BOOL)canSubstituteForClass:(Class)originalObjectClass

Implement to have custom class not be displayed in the inspector of its superclass. Returns YES if self is a valid replacement class for originalObjectClass; NO otherwise.

`connectInspectorClassName`

- (NSString *)connectInspectorClassName

Returns the class name of the receiver's Connection inspector. Interface Builder uses this information to instantiate the Inspector object for the currently selected object. You should rarely need to override the standard Connection inspector.

`editorClassName`

- (NSString *)editorClassName

Returns the class name of the receiver's editor. Interface Builder uses this information to instantiate the editor object for the currently selected object.

helpInspectorClassName

- (NSString *)helpInspectorClassName

Returns the class name of the receiver's Help inspector. Interface Builder uses this information to instantiate the help inspector object for the currently selected object. You should rarely need to override the standard Help inspector.

imageForViewer:

- (UIImage *)imageForViewer

Returns the image that is displayed in the File window when an instance of this class is created. By default, Interface Builder provides an image of a cube. If you want to provide a different image, implement this method in your custom class.

inspectorClassName

- (NSString *)inspectorClassName

Returns the class name of the receiver's Attributes inspector. Interface Builder uses this information to instantiate the attributes inspector object for the currently selected object.

sizeInspectorClassName

- (NSString *)sizeInspectorClassName

Returns the class name of the receiver's size inspector. Interface Builder uses this information to instantiate the size inspector object for the currently selected object.

NSCellAdditions

Characteristic	Description
Category Of:	NSCell
Declared In:	include/InterfaceBuilder/IBViewAdditions.h

Category Description

Interface Builder adds three methods to the definition of the `NSCell` class so that an `NSCell` that is dragged from the Palette window can control its size and make other adjustments as a consequence of resizing.

Instance Methods

`cellWillAltDragWithSize:`

- (void)cellWillAltDragWithSize:(NSSize)cellSize

Allows the cell to set up the necessary state before Alt-dragging.

`maximumSizeForCellSize:`

- (NSSize)maximumSizeForCellSize:(NSSize)cellSize
knobPosition:(IBKnobPosition)knobPosition

Implement this method to control the maximum dimensions of your `NSCell`.

The `knobPosition` argument specifies which control point the user is dragging to resize the object. It can have these values:

```
IBBottomLeftKnobPosition  
IBMiddleLeftKnobPosition  
IBTopLeftKnobPosition  
IBMiddleTopKnobPosition  
IBTopRightKnobPosition  
IBMiddleRightKnobPosition  
IBBottomRightKnobPosition  
IBMiddleBottonKnobPosition
```

You can use the `knobPosition` argument to determine how the `NSCell` will resize.

```
minimumSizeForCellSize:
```

```
- (NSSize)minimumSizeForCellSize:(NSSize)cellSize
knobPosition:(IBKnobPosition)knobPosition
```

Implement this method to control the minimum dimensions of your `NSCell`.

The `knobPosition` argument specifies which control point the user is dragging to resize the object. It can have these values:

```
IBBottomLeftKnobPosition
IBMiddleLeftKnobPosition
IBTopLeftKnobPosition
IBMiddleTopKnobPosition
IBTopRightKnobPosition
IBMiddleRightKnobPosition
IBBottomRightKnobPosition
IBMiddleBottonKnobPosition
```

You can use the `knobPosition` argument to determine how the `NSCell` will resize.

NSView Additions

Characteristic	Description
Category Of:	NSView
Declared In:	include/InterfaceBuilde/IBViewAdditions.h

Category Description

Interface Builder adds four methods to the definition of the `NSView` class so that an `NSView` that is dragged from the Palette window can control its size and make other adjustments as a consequence of resizing. As the user begins to drag one of the `NSView`'s control points, Interface Builder sends it a `maximumSizeFromKnobPosition:` or `minimumSizeFromKnobPosition:` message. When the user releases the mouse button, the View receives a `placeView:` message.

Instance Methods

`allowsAltDragging:`

- (BOOL)allowsAltDragging

Return YES to allow your `NSView` subclass to be Alt-dragged to create a matrix. The `NSView` must provide a cell for the resulting matrix.

`maximumSizeFromKnobPosition:`

- (NSSize)maximumSizeFromKnobPosition:(IBKnobPosition)knobPosition

Implement this method to control the maximum dimensions of your `NSView`.

The `knobPosition` argument specifies which control point the user is dragging to resize the object. It can have these values:

```
IBBottomLeftKnobPosition
IBMiddleLeftKnobPosition
IBTopLeftKnobPosition
IBMiddleTopKnobPosition
IBTopRightKnobPosition
IBMiddleRightKnobPosition
IBBottomRightKnobPosition
IBMiddleBottomKnobPosition
```

You can use the `knobPosition` argument to determine how the `NSView` will resize.

`minimumSizeFromKnobPosition:`

- (NSSize)minimumSizeFromKnobPosition:(IBKnobPosition)knobPosition

Implement this method to control the minimum dimensions of your `NSView`.

The `knobPosition` argument specifies which control point the user is dragging to resize the object. It can have these values:

```
IBBottomLeftKnobPosition
IBMiddleLeftKnobPosition
IBTopLeftKnobPosition
IBMiddleTopKnobPosition
```

```
IBTopRightKnobPosition
IBMiddleRightKnobPosition
IBBottomRightKnobPosition
IBMiddleBottonKnobPosition
```

You can use the `knobPosition` argument to determine how the `NSView` will resize. For example, an `NSBox` determines which control point is being dragged and then lets the user shrink its size from that point only to the degree that no subview (`NSButton`, `NSTextField`) would be obscured.

`placeView:`

```
- (void)placeView:(NSRect)newFrame
```

Notifies an `NSView` of a change in its frame size. Interface Builder's implementation of this method is to send a `setFrame:` message to the receiver, using `newFrame` as the argument.

You can implement this method, for example, to resize the `NSView`'s subviews. In your implementation, you should also send a `setFrame:` message to `self` to set the `NSView`'s new size.

Interface Builder API Protocols



IB

Characteristic	Description
Adopted By:	no OpenStep classes
Declared In:	include/InterfaceBuilder/IBApplicationAdditions.h

Protocol Description

Interface Builder's subclass of the `NSApplication` class conforms to the `IB` protocol. Thus, objects in your custom palette can interact with Interface Builder's main module by sending messages (corresponding to the methods in this protocol) to the `NSApplication` object `NSApp`. for your application.

Method Types

Activity	Class Methods
Accessing the document	- activeDocument
Accessing the selection owner	- selectionOwner
Querying the mode	- isTestingInterface

Instance Methods

`activeDocument`

- (id<IBDocuments>)activeDocument

Returns the active document, as represented by an object that conforms to the `IBDocuments` protocol. The active document is represented by the nib file window of the document containing the selection.

`isTestingInterface`

- (BOOL)isTestingInterface

Returns YES if Interface Builder is in Test mode.

`selectionOwner`

- (id<IBSelectionOwners>)selectionOwner

Returns the owner of the currently selected object or `nil` if no object is selected.

IBConnectors

Characteristic	Description
Adopted By:	no OpenStep classes
Declared In:	include/InterfaceBuilder/IBConnectors.h

Protocol Description

This protocol declares the programmatic interface of connector objects. Connectors are designed to store information about connections between objects in a nib document. For example, the private class that Interface Builder uses to store information about outlet connections conforms to this protocol and adds a method to store the name of the outlet. Connector objects are archived in the nib file.

When an application loads a nib file, the connector objects in the nib file are unarchived and are sent the `establishConnection` message. In response to this message, the connector establishes its type of connection between the source and destination. For example, Interface Builder's outlet connector sets the named outlet to the destination object.

Your Connection inspector must set the source and destination in each of its connectors (for example, with `setSource:` and `setDestination:` methods). The protocol does not include methods to set these outlets, only to query them.

Method Types

Activity	Class Methods
Establishing a connection	– establishConnection
Labeling a connection	– label
Querying outlets	– destination – source – nibInstantiate
Updating a connector	– replaceObject:withObject

Instance Methods

`destination`

– (id)destination

Implement to return the object that is the destination of the connection. See also
– `source`.

`establishConnection`

– (void)establishConnection

Implement to connect the source and destination objects. Interface Builder sends this message to each connector object after all objects have been unarchived from the nib file.

`label`

– (NSString *)label

Implement to label the connection when it is displayed in the nib file window in outline mode.

nibInstantiate

```
- (id)nibInstantiate
```

Implement to verify the identities of the connector's source and destination objects.

Interface Builder sends a `nibInstantiate` message to allow a connector object to verify that its `source` and `destination` instance variables point to the intended objects. For example, consider the case in which a user puts an `NSCustomView` in a window and then reassigns the `NSCustomView`'s class to `MyView`. The `MyView` class has a `textfield` outlet that the user connects to a neighboring `NSTextField` object. This action causes Interface Builder to create a connector object and set its destination to the `NSTextField` and its source to the `NSCustomView`. The source cannot be set to the `MyView` object since that class does not exist in InterfaceBuilder, which is why the `NSCustomView` was used.

When the resulting nib file is loaded in the finished application, the connector object is unarchived and sent a `nibInstantiate` message. At this point the connector must reset its `source` instance variable from the `NSCustomView` object to the `MyView` object.

The Application Kit, in a category of `NSObject`, provides a default implementation of `nibInstantiate`. This implementation returns `self`. Consequently, all objects can respond to a `nibInstantiate` message. Your connector, therefore, should minimally implement `nibInstantiate` messages to transmit this message to its source and destination objects. For example, assuming the outlets are named `theSource` and `theDestination`, the implementation is as follows:

```
- (id)nibInstantiate
{
    theSource = [theSource nibInstantiate];
    theDestination = [theDestination nibInstantiate];
    return self;
}
```

This allows the source and destination objects to return the `ids` of the intended objects.

`replaceObject:withObject:`

- (void)replaceObject:(id)oldObject withObject:(id)newObject

Implement to update a connector by replacing its old source or destination object (`oldObject`) with a new object (`newObject`). This is used by Interface Builder, for example, when a user drags an `NSButton` object into an `NSMatrix` of `NSButtonCells`. Assuming that the `NSButton` was connected, the connection information must be updated to reflect the replacement of the `NSButton` by an `NSButtonCell`. Interface Builder accomplishes this by sending the appropriate connector object a `replaceObject:withObject:` message with the `NSButton` as `oldObject` and the `NSButtonCell` as `newObject`.

`source`

- (id)source

Implement to return the object that is the source of the connection. See also `destination`.

IBDocuments

Characteristic	Description
Adopted By:	no OpenStep classes
Declared In:	include/InterfaceBuilder/IBDocuments.h

Protocol Description

`IBDocuments` is the protocol to use to communicate with Interface Builder's document object. The document object is private to Interface Builder but may be accessed by sending Interface Builder's subclass of `NSApplication` an `activeDocument` message:

```
theActiveDoc = [NSApp activeDocument];
```

The document object maintains the following components of a document:

- The object hierarchy
- The list of connectors
- The active editor

It also mediates in copy and paste operations and controls the redisplay of objects in Interface Builder.

Through the document object, you keep Interface Builder informed of changes to the data structure that you want archived in the nib file. For example, if your custom editor allows the user to add an object by dragging it into the editor's window, you must inform Interface Builder of this addition by sending the document object an `attachObject:toParent:` message. Interface Builder does not archive an object in the nib file unless it has been added to the object hierarchy.

Note – A paste operation, which uses the `pasteType:fromPasteboard:parent:` method, automatically updates the hierarchy.)

Method Types

Activity	Class Method
Managing the document	<ul style="list-style-type: none"> – touch – documentPath:
Managing the object hierarchy	<ul style="list-style-type: none"> – attachObject:toParent: – attachObjects:toParent: – detachObject – detachObjects – replaceObject:withObject: – objects: – containsObject: – parentOfObject: – copyObject:type:toPasteboard: – copyObjects:type:toPasteboard: – pasteType:fromPasteboard:parent::
Setting object names	<ul style="list-style-type: none"> – setName:forObject: – nameForObject: – containsObjectWithName:forParent:

Activity	Class Method
Managing connectors	<ul style="list-style-type: none">- addConnector:- removeConnector:- connectorsForSource:- connectorsForSource:ofClass:- connectorsForDestination:- connectorsForDestination:ofClass
Managing editors	<ul style="list-style-type: none">- openEditorForObject:- editorForObject:create:- setSelectionFromEditor:- resignSelectionForEditor:- editor:didCloseForObject:
Updating the display	<ul style="list-style-type: none">- drawObject:

Instance Methods

`addConnector:`

- (void)addConnector:(id<IBConnectors>)connector

Adds a connector object to the list maintained by the document. (See “IBConnectors” on page D-3 for more information on connectors.) This is the message a custom connection inspector sends Interface Builder’s document object to register a connection. See also - removeConnector:.

`attachObject:toParent:`

- (void)attachObject:(id)object toParent:(id)parent

Adds `object` to the document’s object hierarchy by attaching it to `parent`. This method (and the related method `attachObjects:toParent:`) lets you keep the document’s object hierarchy informed of changes in the objects under the control of your custom editor. See also - attachObjects:toParent:.

`attachObjects:toParent:`

`-(void)attachObjects:(NSArray *)objects toParent:(id)parent`

Adds the objects in `objects` to the document's object hierarchy by attaching them to `parent`. This method (and the related method `attachObject:toParent:`) lets you keep the document's object hierarchy informed of changes in the objects under the control of your custom editor. See also `- attachObject:toParent:`.

`connectorsForDestination:`

`-(NSArray*)connectorsForDestination:(id)destination`

Places all connector objects whose destinations are `destination` in an array and returns the array.

An object can be the destination of multiple connections, so the connectors information are returned in an array. Each connector in the array conforms to the `IBConnectors` protocol and contains the information for one connection.

Do not free the connection objects ; the array will autorelease. See also

`- connectorsForDestination:ofClass:`, `- connectorsForSource:`.

`connectorsForDestination:ofClass:`

`-(NSArray*)connectorsForDestination:(id)destination
ofClass:(id)classObject`

Places the connector objects of class `classObject` whose destinations are `destination` in an array and returns the array.

An object can be the destination of multiple connections, so the connectors are returned in an array. Each connector in the array conforms to the `IBConnectors` protocol and contains the information for one connection.

Do not free the connection objects ; the array will autorelease. See also

`- connectorsForDestination:`, `- connectorsForSource:`.

`connectorsForSource:`

`-(NSArray*)connectorsForSource:(id)source`

Places all connector objects whose sources are `source` in an array and returns the array.

A source can have multiple connections, so the connectors are returned in an array. Each connector in the array conforms to the `IBConnectors` protocol and contains the connection information for one connection.

Do not free the connection objects ; the array will autorelease. See also

`- connectorsForSource:ofClass:`, `- connectorsForDestination:`.

`connectorsForSource:ofClass:`

`-(NSArray*)connectorsForSource:(id)source
ofClass:(id)classObject`

Places the connector objects of class `classObject` whose sources are `source` in an array and returns the array.

A source can have multiple connections, so the connectors are returned in an array. Each connector in the array conforms to the `IBConnectors` protocol and contains the connection information for one connection.

Do not free the connection objects ; the array will autorelease. See also

`- connectorsForSource:`, `- connectorsForDestination:`.

`containsObject:`

`-(BOOL)containsObject:(id)object`

Returns YES if `object` is a part of the document's object hierarchy; NO otherwise. You might send a `containsObject:` message to the document object before attempting to open a subeditor for `object`.

`containsObjectWithName:forParent`

```
- (BOOL)containsObjectWithName:(NSString *)name  
forParent:(id)parent
```

Returns YES if the document contains an object named name that has the parent parent.

`copyObject:type:toPasteboard:`

```
- (BOOL)copyObject:(id)object  
type:(NSString*)type  
toPasteboard:(NSPasteboard *)pasteboard
```

Copies object to the specified pasteboard. The type argument can be one of the following:

```
IBObjectPboardType  
IBCellPboardType  
IBMenuPboardType  
IBMenuCellPboardType  
IBViewPboardType  
IBWindowPboardType
```

An editor should send the active document object a `copyObject:type:toPasteboard:` or `copyObjects:type:toPasteboard:` message as part of its implementation of its `copySelection` method. See also `copyObjects:type:toPasteboard:.`

`copyObjects:type:toPasteboard:`

```
- (BOOL)copyObjects:(NSArray*)objects  
type:(NSString*)type  
toPasteboard:(NSPasteboard *)pasteboard
```

Copies the objects in objects to the specified pasteboard. The type argument can be one of the following:

```
IBObjectPboardType  
IBCellPboardType  
IBMenuPboardType
```

IBMenuCellPboardType
IBViewPboardType
IBWindowPboardType

An editor should send the document object a `copyObject:type:toPasteboard:` or `copyObjects:type:toPasteboard:` message as part of its implementation of its `copySelection` method. See also

- `copyObject:type:toPasteboard:.`

`detachObject:`

- (void)detachObject:(id)object

Removes `object` from the document's object hierarchy. An editor should send the document object a `detachObject:` or `detachObjects:` message as part of its implementation of the `deleteSelection` method. If any connections are associated with the object, they are deleted as well. See also

- `detachObjects:.`

`detachObjects:`

- (void)detachObjects:(NSArray *)objects

Removes the objects in `objects` from the document's object hierarchy. An editor should send the document object a `detachObject:` or `detachObjects:` message as part of its implementation of the `deleteSelection` method. If any connections are associated with the object, they are deleted as well. See also - `detachObject:.`

`documentPath`

- (NSString *)documentPath

Returns the path name for the document's directory wrapper. The path is displayed as the title of Interface Builder's File window.

`drawObject:`

- (void)drawObject:(id)object

Redraws the selected object by opening its editor, the editor for its parent object, and so on up the object hierarchy. Each editor is also sent a `resetObject:` message.

`editor:didCloseForObject:`

- (void)editor:(id<IBEditors>)editor didCloseForObject:(id)object

Informs the document object that `anEditor` is no longer active. By sending this message to the document object when an editor is closed, Interface Builder's record of the active editor is kept up to date. Interface Builder itself invokes this method whenever an editor is closed because of a user action, such as the closing of a window.

`editorForObject:create:`

- (id<IBEditors>)editorForObject:(id)object create:(BOOL)createIt:

Returns the editor object for `object`. If `createIt` is YES and the editor has not been instantiated, it is instantiated and returned. If `createIt` is NO, the editor is returned only if it has already been instantiated. If `createIt` is NO and the editor has not been instantiated, this method returns `nil`.

`nameForObject:`

- (NSString*)nameForObject:(id)object

Returns the name associated with `object`. See also - `setName:forObject:.`

`objects:`

- (NSArray*)objects:

Returns the objects from the document's object hierarchy in an array. The objects are not arranged in any particular order.

`openEditorForObject:`

`– (id<IBEditors>)openEditorForObject:(id)object`

Opens the editor object for `object`. This method ensures that editors for all the objects above `object` in the object hierarchy are open before opening `object`'s editor.

`parentOfObject:`

`– (id)parentOfObject:(id)object`

Returns the object above `object` in the document's object hierarchy. The top object is the File's owner. Returns `nil` if `object` is the File's owner.

`pasteType:fromPasteboard:parent:`

`– (NSArray*)pasteType:(NSString *)type
fromPasteboard:(NSPasteboard *)pasteboard
parent:(id)parent`

Returns an array containing the objects that are in the pasteboard. The pasteboard and the type being pasted are identified by `pasteboard` and `type`.

The implementation of this method invokes `attachObjects:toParent:` and `touch`. The returned array object provides your code with access to pasted objects so you can add them to your code's data structures. The array will autorelease.

`removeConnector:`

`– (void)removeConnector:(id<IBConnectors>)connector`

Removes `connector` from the list of connectors maintained by the document. (See "IBConnectors" on page D-3 for more information on connectors.) A custom connection inspector sends the document object this message to break a connection.

Interface Builder does not free `connector`; it is your responsibility to do so. See also `– addConnector:`.

replaceObject:withObject:

```
- (void)replaceObject:(id)oldObject withObject:(id)newObject
```

Implement to update a connector by replacing its old source or destination object (oldObject) with a new object (newObject). For example, this is used by Interface Builder when a user Alt-drags a resize handle on an NSButton object causing the NSButton to be replaced by an NSMatrix of NSButtonCells. Assuming that the NSButton was connected, the connection information must be updated to reflect that fact that the NSButton has been replaced by an NSButtonCell. Interface Builder updates this information by sending the appropriate connector object a replaceObject:withObject: message with the NSButton as oldObject and the NSButtonCell as newObject.

resignSelectionForEditor:

```
- (void)resignSelectionForEditor:(id <IBEditors>)editor
```

Unregisters editor as the editor that owns the selection. See also
- setSelectedFromEditor:.

setName:forObject:

```
- (BOOL)setName:(NSString*)name forObject:(id)object
```

Sets the name associated with object. For objects in the File window, this is the name displayed below the object's image. Except for objects in the File window, setting an object's name is generally not needed. See also

```
- nameForObject:.
```

setSelectedFromEditor:

```
- (void)setSelectedFromEditor:(id<IBEditors>)editor
```

Registers editor as the editor that owns the selection.

When you activate an editor or change the selection, make sure you send this message to the document object to keep Interface Builder informed of the selection's owner. In this way, when the user switches from one window to another, or from one document to another, Interface Builder can inform the proper editor to display its selection. Also, Interface Builder uses the selection information to determine which inspector to display in its Inspector panel.

touch

- (void)touch

Marks the document as edited. When you deactivate an editor, the nib file window's close box displays a broken "X" to indicate the edited status.

IBEditors

Characteristic	Description
Adopted By:	no OpenStep classes
Incorporates:	IBSelectionOwners
Declared In:	include/InterfaceBuilder/IBEditors.h

Protocol Description

IBEditors, and the IBSelectionOwners protocol that it incorporates, define the required programmatic interface to an editor object in Interface Builder. When a user double-clicks on a custom object, Interface Builder instantiates the object's editor (using initWithObject:inDocument:). (Interface Builder would have previously determined the editor's class by sending the custom object a getEditorClassName message. See "NSObject Additions" on page C-11 for more information.) The editor presents its window, allowing the user to make alterations to the displayed data.

For example, assume that a custom palette provides an AddressBook object. Once instantiated in the File window, the AddressBook object can be double-clicked to activate the editor. The editor presents the user with a window that permits the entry of names and addresses. As data is entered, the editor can update the AddressBook object with the new information.

Besides letting users edit an object's state, an editor intercedes in copy and paste operations. When the user chooses the Cut or Copy command, Interface Builder sends a deleteSelection or copySelection message to the editor. The editor takes the appropriate action and then alerts Interface Builder's document object that the cut or copy operation has occurred. This keeps the document object up-to-date with the actual state of the document.

When a paste operation is attempted, Interface Builder sends the active editor an `acceptsTypeFromArray:` message to determine if it will accept any of the types on the pasteboard. If the editor refuses the offered types, Interface Builder sends the same message to the next higher editor in the object hierarchy, and so on until it reaches the top. This explains why, if a paste operation is attempted when an `NSButton` object is on the pasteboard and the Pop-up list editor is open, nothing is pasted in the selected `NSPopUpList`; instead, the `NSButton` is pasted in the window that contains the `NSPopUpList`. The `NSPopUpList` refused the pasteboard type, but the view editor accepted it.

If the editor accepts one of the offered types, the editor receives a `pasteInSelection` message. The editor then replaces the selection with the pasted data and alerts Interface Builder of the change by sending the document object a `pasteType:fromPasteboard:parent:` message.

Editors also control the opening and closing of subeditors.

Method Types

Activity	Class Methods
Editing objects	- initWithObject:inDocument: - close:
Identifying objects	- document - editedObject - window
Displaying objects	- resetObject:
Managing the selection	- wantsSelection - selectObjects: - makeSelectionVisible:
Copying and pasting objects	- copySelection - deleteSelection - pasteInSelection - acceptsTypeFromArray:
Opening and closing editors	- openSubeditorForObject: - closeSubeditors
Activating the editor	- orderFront - activate - validateEditing:

Instance Methods

acceptsTypeFromArray:

- (BOOL)acceptsTypeFromArray:(NSArray*)types

Implement to return the pasteboard types your editor accepts. `types` is an array of strings holding the type names. If your editor does not accept any of the supplied types, it should return `nil`.

For example, if an editor only accepts the type `IBObjectPboardType`, it could implement this method in the following way:

```
- (BOOL)acceptsTypeFromArray:(NSArray*)types
{
    int i = 0;
    if (!types) {
        return nil;
    }
    for (i=0; i<[types count];i++){
        if ([[types objectAtIndex:i] isEqual:IBObjectPboardType]){
            return IBObjectPboardType;
        }
    }
    return nil;
}
```

activate

- (BOOL)activate

Implement to activate the editor. Typically, an editor activates itself by making its window key, displaying its selection, and advising the document object that it owns the selection:

```
- (BOOL)activate
{
    [window makeKeyAndOrderFront:self];
    [self makeSelectionVisible:YES];
    [document setSelectionFrom:self];
    return YES;
}
```

Your implementation of this method should return YES if the editor activates itself and NO otherwise.

When a user double-clicks on an object controlled by an editor, the editor receives an `orderFront` and then an `activate` message. See also

- `orderFront`.

`close`

- (void)close

Implement to close the editor and free its resources. This method can be invoked for a number of reasons. For example, Interface Builder invokes this method when the user closes the document. Or, your editor might send itself a `close` message when the user closes the editor's window.

As part of the implementation of this method, send an `editorDidClose:forObject:` message to the active document to inform IB that the editor has closed:

```
[[NSApp activeDocument] editorDidClose:self  
forObject:editedObject];
```

See also - `editorDidClose:forObject:` (IBDocuments protocol).

`closeSubeditors`

- (void)closeSubeditors

Implement to close all subeditors. See also - `openSubeditorForObject:`.

`copySelection`

- (BOOL)copySelection

Implement to copy the selected object(s) to the pasteboard. When the user chooses the Cut or Copy commands in Interface Builder, the editor that owns the selection receives a `copySelection` message.

In your implementation of this method, you should send the document object a `copyObject:type:inPasteboard:` or a `copyObjects:type:inPasteboard:` message, as declared in the IBDocuments protocol. Return YES if the selection was copied to the pasteboard; NO otherwise. See also - `deleteSelection`.

`deleteSelection`

- (BOOL)deleteSelection

Implement to delete the selected object(s). This method is invoked when the user deletes the selection by using the Delete key, the Delete command, or as part of the Cut command (after the selection has been copied using the `copySelection` method).

In your implementation of this method, you should send the document object a `detachObject:` or a `detachObjects:` message, as declared in the `IBDocuments` protocol. Return YES if the selection was deleted; NO otherwise. See also - `copySelection`.

`document`

- (id<IBDocuments>)document

Implement this method to return the document to which the object being edited belongs.

`editedObject`

- (id)editedObject

Implement to return the object that is being edited. This is generally the object on which the user double-clicked to open the editor.

`initWithObject:inDocument:`

- (id)initWithObject:(id)object
inDocument:(id/*<IBDocuments>*/)document

Implement this method to initialize a newly allocated editor. `object` is the object that is being edited (for example, the object on which the user has double-clicked). `document` is the currently active document, as would be returned by sending an `activeDocument` message to `NSApp`. Typically, an editor object caches the document object in one of its instance variables, since editors must frequently communicate with the document object.

`makeSelectionVisible:`

- (void)makeSelectionVisible:(BOOL)showIt

Implement to add or remove the selection markings from the current selection. An editor receives a `makeSelectionVisible:` message whenever Interface Builder wants to ensure that the selection is properly marked. For example, when a window becomes key, the editor that owns the selection in the window receives a `makeSelectionVisible:YES` message. When the window loses its key window status, the editor that owns the selection receives a `makeSelectionVisible:NO` message.

`openSubeditorForObject:`

- (id<IBEditors>)openSubeditorForObject:(id)object

Implement to open the subeditor for anObject. An editor receives this message when the user double-clicks in the editor's selection. For the return value of this method, the editor should return `nil` if there is no subeditor; otherwise, it should return of the subeditor object.

`orderFront`

- (void)orderFront

Implement to bring the editor's window to the front. When a user double-clicks on an object, the controlling editor receives an `orderFront` and then an `activate` message. See also - `activate`.

`pasteInSelection`

- (BOOL)pasteInSelection

Implement to paste the object(s) from the pasteboard into the current selection. When the user chooses the Paste command in Interface Builder, the editor that owns the selection receives a `pasteInSelection` message. The implementation of the corresponding method should invoke the document object's `pasteType:fromPasteboard:parent:` method.

This method should return `YES` if the paste operation was successful; `NO` otherwise. See also - `pasteType:fromPasteboard:parent:` (`IBDocuments` protocol).

resetObject:

- (void)resetObject:(id)object

This method should redraw `object`. When the document object receives a `drawObject:` message, it makes sure that the editor for that object—and for each of its parent objects—is open. It then sends `resetObject:` messages to each of the editors in this object hierarchy.

selectObjects:

- (void)selectObjects:(NSArray *)objects

Implement to draw the objects for the array to indicate that they are all selected.

validateEditing

- (void)validateEditing

Causes the value of the selection to be set to the value of the field being edited, if any. “Being edited” does not necessarily mean a user is typing. If a field (for example, an `NSTextField` object) has Interface Builder’s global `NSText` object in its place as first responder, then the field is considered as being edited.

wantsSelection

- (BOOL)wantsSelection

Implement to return `YES` if the editor is willing to become the selection owner; `NO` if not.

window

- (NSWindow*)window

Implement to return the editor’s window.

IBSelectionOwners

Characteristic	Description
Adopted By:	no OpenStep classes
Declared In:	include/InterfaceBuilder/IBEditors.h

Protocol Description

All editors must conform to the `IBSelectionOwners` protocol. By implementing this protocol, an editor advertises its selection to other objects in Interface Builder. (The *selection* is that object or objects that would be copied if the user chose the Copy command.)

For example, Interface Builder invokes an editor's `selectionCount` and `getSelectionInto:` methods to determine how to update the Inspector panel. If the selection count is more than one, the Inspector panel displays the message "Multiple Selection". If there is only one object in the selection, Interface Builder invokes the editor's `getSelectionInto:` method to access the object and then determines the appropriate inspector to display in the Inspector panel.

Instance Methods

`drawSelection`

- (void)drawSelection

Implement this method to redraw the objects in the selection.

`selection:`

- (NSArray*)selection

Implement this method to place the currently selected objects into an array object. If the editor does not have a selection, it should return an empty array. See also - `selectionCount`.

selectionCount

- (unsigned int)selectionCount

Implement to return the number of objects in your editor's selection. See also

- selection:.

Interface Builder API Types and Constants



Symbolic Constants

Control Point Constants

Characteristic	Description
Declared In:	include/InterfaceBuilder/IBViewAdditions.h

Synopsis

IB_BottomLeftKnobPosition
IB_MiddleLeftKnobPosition
IB_TopLeftKnobPosition
IB_MiddleTopKnobPosition
IB_TopRightKnobPosition
IB_MiddleRightKnobPosition
IB_BottomRightKnobPosition
IB_MiddleBottomKnobPosition

Description

These constants identify the control points that appear around a selected `NSView` object in an application that is under construction. See the descriptions of the `minimumSizeForCellSize:` and `maximumSizeForCellSize:` methods in “`NSCellAdditions`” on page C-14 and the

`minimumSizeFromKnobPosition:` and `maximumSizeFromKnobPosition:` methods in “*NSView Additions*” on page C-15 for more information.

Global Variables

Notification Types

Characteristic	Description
Declared In:	include/InterfaceBuilder/IBConnectors.h

Synopsis

```
NSString IBWillAddConnectorNotification
NSString IBDidAddConnectorNotification
NSString IBWillRemoveConnectorNotification
NSString IBDidRemoveConnectorNotification
NSString IBDidOpenDocumentNotification
NSString IBWillSaveDocumentNotification
NSString IBDidSaveDocumentNotification
NSString IBWillCloseDocumentNotification
```

Description

These global variables identify additional notification types used by Interface Builder. The types are used in notifying when Interface Builder documents are opened or saved, or when connections between objects have been added or removed.

Characteristic	Description
Declared In:	include/InterfaceBuilder/IBEditors.h

Synopsis

```
NSString IBSelectionChangedNotification  
NSString IBAttributesChangedNotification
```

Description

These global variables identify some additional notification types used by Interface Builder. The types are used in notifying when the selection has changed and when the user has changed an object's attributes in the Inspector.

Pasteboard Types

Characteristic	Description
Declared In:	include/InterfaceBuilder/IBPalette.h

Synopsis

```
NSString IBOBJECTPBOARDTYPE ;  
NSString IBCELLPBOARDTYPE ;  
NSString IBMENUPBOARDTYPE ;  
NSString IBMENUCELLPBOARDTYPE ;  
NSString IBVIEWPBOARDTYPE ;  
NSString IBWINDOWPBOARDTYPE ;
```

Description

These global variables identify some additional pasteboard types used by Interface Builder. See “IBPalette” on page C-6 for information on the use of these types.

Copyright 1996 Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, Californie 94043-1100 USA.

Tous droits réservés. Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, et la décompilation. Aucune partie de ce produit ou de sa documentation associée ne peuvent être reproduits sous aucune forme, par quelque moyen que ce soit sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il en a.

Des parties de ce produit pourront être dérivées du système UNIX[®], licencié par UNIX Systems Laboratories, Inc., filiale entièrement détenue par Novell, Inc., ainsi que par le système 4.3. de Berkeley, licencié par l'Université de Californie. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

LEGENDE RELATIVE AUX DROITS RESTREINTS: l'utilisation, la duplication ou la divulgation par l'administration américaine sont soumises aux restrictions visées à l'alinéa (c)(1)(ii) de la clause relative aux droits des données techniques et aux logiciels informatiques du DFAR 252.227- 7013 et FAR 52.227-19.

Le produit décrit dans ce manuel peut être protégé par un ou plusieurs brevet(s) américain(s), étranger(s) ou par des demandes en cours d'enregistrement.

MARQUES

Sun, Sun Microsystems, le logo Sun, SunSoft, le logo SunSoft, Solaris, SunOS, OpenWindows, DeskSet, ONC, ONC+, NFS, et NEO sont des marques déposées ou enregistrées par Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. OpenStep est une marque enregistrée de NeXT Software, Inc. UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays, et exclusivement licenciée par X/Open Company Ltd. OPEN LOOK est une marque enregistrée de Novell, Inc. PostScript et Display PostScript sont des marques d'Adobe Systems, Inc. Object Design est une marque déposée et le logo Object Design est une marque enregistrée d'Object Design, Inc.

Toutes les marques SPARC sont des marques déposées ou enregistrées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. SPARCcenter, SPARCcluster, SPARCcompiler, SPARCdesign, SPARC811, SPARCengine, SPARCprinter, SPARCserver, SPARCstation, SPARCstorage, SPARCworks, microSPARC, microSPARC-II et UltraSPARC sont exclusivement licenciées à Sun Microsystems, Inc. Les produits portant les marques sont basés sur une architecture développée par Sun Microsystems, Inc.

Les utilisateurs d'interfaces graphiques OPEN LOOK[®] et Sun[™] ont été développés par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique, cette licence couvrant aussi les licenciés de Sun qui mettent en place OPEN LOOK GUIs et qui en outre se conforment aux licences écrites de Sun.

Le système X Window est un produit du X Consortium, Inc.

Ce produit incorpore la technologie licenciée par Object Design, Inc.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" SANS GARANTIE D'AUCUNE SORTE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS A REPOUDRE A UNE UTILISATION PARTICULIERE OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.

CETTE PUBLICATION PEUT CONTENIR DES MENTIONS TECHNIQUES ERRONEES OU DES ERREURS TYPOGRAPHIQUES. DES CHANGEMENTS SONT PERIODIQUEMENT APPORTES AUX INFORMATIONS CONTENUES AUX PRESENTES. CES CHANGEMENTS SERONT INCORPORES AUX NOUVELLES EDITIONS DE LA PUBLICATION. SUN MICROSYSTEMS INC. PEUT REALISER DES AMELIORATIONS ET/OU DES CHANGEMENTS DANS LE(S) PRODUIT(S) ET/OU LE(S) PROGRAMME(S) DECRITS DANS DETTE PUBLICATION A TOUS MOMENTS.

Index

A

- abstract classes 8-12
- **acceptsTypeFromArray:** (IBEditors) D-18
- action 3-5, 3-91, 3-111, 3-112, 3-113
 - adding to class 3-145
 - allowing 3-51
 - and First Responder object 3-20
 - connecting 3-116, 3-120, 3-151
 - File's Owner as target of 3-19
 - mouse 3-52
 - naming 3-146
 - reducing 3-52
 - sent by button click 3-72
 - specifying 3-143
 - specifying for class 3-143
 - target of 3-91
 - within panel 3-71
 - without explicit target 3-20
- action message. See action
- action messages 8-37
- action method. See action
- **activate** (IBEditors) D-18
- **activeDocument** (IB) D-2
- Add action command in Interface Builder 3-146
- Add command in Project Builder 2-34
- Add Help Directory command in Project Builder 2-33, 3-134
- Add outlet command in Interface Builder 3-144
- **addConnector:** (IBDocuments) D-8
- adding existing classes 3-162
- Align Left command in Edit 4-29
- Align Right command in Edit 4-29
- Align submenu in Interface Builder 3-175
- Align To Grid command in Interface Builder 3-47, 3-176
- Alignment command in Interface Builder 3-45, 3-177
- alignment grid 3-45
- alignment in Edit
 - paragraph 4-29
- alignment of text 3-84
- Alignment panel in Interface Builder 3-46
- + **alloc** method 8-16
- allocating instances 8-16
- **allowsAltDragging** (NSView Additions) C-16
- application
 - building in Interface Builder 3-4
 - debugging 2-32, A-1–A-20
 - directory 1-6
 - installation 1-6

application (*Continued*)
 project in Project Builder 2-2
 running in Project Builder 2-32
 search path 1-6
Application Kit 3-4, 3-5, 3-141
arguments, variable number of 8-5, 8-23,
 8-26
arranging objects 3-45
assigning Command key 3-61
– **associateObject:ofType:withView:**
 (IBPalette) C-8
– **attachObject:toParent:** (IBDocuments)
 D-8
– **attachObjects:toParent:** (IBDocuments)
 D-9
attributes
 automatic resizing 3-99, 3-100
 examining 3-65
 images 3-80
 of images in buttons 3-77
 of NSBox 3-85
 of NSBrowser 3-87
 of NSButton 3-71
 of NSMatrix 3-95
 of NSMenuItem 3-89
 of NSScrollView 3-82
 of NSTextField 3-81
 of panel 3-67
 of pop-up/pull-down lists 3-89
 of sounds in buttons 3-77
 of text 3-84
 of window 3-67
 sounds 3-80
Attributes display 3-65
 and NSView subclass 3-160
automatic resizing 3-98
 examples 3-102
 potential conflicts 3-101
autotyping of outlets 3-166
auxiliary nib file 3-19, 3-23, 3-137
– **awakeFromDocument** (NSObject
 Additions) C-12

B

backtrace
 setting breakpoint for A-18
branch node 3-87
breaking connections 3-130
breakpoint
 setting for backtrace A-18
 setting in Objective C method A-2
 stopping at when filter matches A-13,
 A-15, A-16, A-17
 to catch errors A-5
Bring To Front command in Interface
 Builder 3-57
browser 3-87
 attributes 3-87
 structure 3-93
buffered window 3-69
Build Application command in Project
 Builder 2-34
bundle project in Project Builder 2-2
button
 and sounds and images 3-75
 attributes 3-71
 icon 3-71
 methods 3-73
 options 3-74
 radio 3-71, 3-96
 setting key equivalent 3-73
 structure 3-72
 switch 3-71, 3-96
 title 3-73
 type 3-74

C

C++, using with Objective C 8-1
+ **canSubstituteForClass** (NSObject
 Additions) C-12
cell options (matrix) 3-98
cell prototype (matrix) 3-96
– **cellWillAltDragWithSize** (NSCell
 Additions) C-14
Center command in Edit 4-29

-
- Check Spelling command in Edit 4-26
 - class
 - and Attributes display 3-66
 - and connecting actions 3-151
 - and connecting outlets 3-149
 - creating 3-137–3-164
 - creating instance of 3-147
 - locating 3-143
 - naming 3-141
 - selecting superclass 3-141
 - updating definition 3-163
 - Class data type 8-15
 - @class** directive 8-24
 - class hierarchy 3-142
 - viewing 3-143
 - + **class** method 8-15
 - **class** method 8-15
 - class methods 8-15, 8-22
 - class objects 8-8, 8-14–8-19, 8-20
 - classes 8-8–8-31
 - declaration of 8-21–8-25
 - implementation of 8-25–8-31
 - Classes display in Interface Builder 3-14, 3-15, 3-141, 3-142, 3-143
 - Close Ancestors command in Edit 4-34
 - close button 3-69
 - Close Descendants command in Edit 4-34
 - **close** (IBEditors) D-19
 - **closeSubeditors** (IBEditors) D-19
 - _cmd** 8-39
 - code
 - generating 3-153
 - columns of objects 3-48
 - Command command in Edit 4-22, 4-32
 - Command key
 - assigning 3-61
 - connecting
 - actions 3-151
 - objects 3-113
 - outlets 3-149
 - **connectInspectorClassName** (NSObject Additions) C-12
 - connections
 - and outline mode 3-13
 - between fields 3-128
 - examining 3-122
 - in outline mode 3-120
 - within interface 3-118
 - Connections display 3-114, 3-117, 3-120, 3-122, 3-123, 3-124, 3-128, 3-130, 3-149, 3-152, 3-156
 - **connectorsForDestination:** (IBDocuments) D-9
 - **connectorsForDestination:ofClass:** (IBDocuments) D-9
 - **connectorsForSource:** (IBDocuments) D-10
 - **connectorsForSource:ofClass:** (IBDocuments) D-10
 - consistency
 - and UI design 3-51
 - **containsObject:** (IBDocuments) D-10
 - **containsObjectWithName:forParent:** (IBDocuments) D-11
 - contents of nib file 3-24
 - Contract All command in Edit 4-31
 - Contract Sel command in Edit 4-31
 - controller object 3-114, 3-117
 - coordinate system 3-50
 - Copy command in Interface Builder 3-44
 - Copy PS command in Edit 4-34
 - Copy Ruler command in Edit 4-30
 - copying objects
 - between nib files 3-44
 - between windows 3-44
 - **copyObject:type:toPasteboard:** (IBDocuments) D-11
 - **copyObjects:type:toPasteboard:** (IBDocuments) D-11
 - **copySelection** (IBEditors) D-19
 - creating a nib file 3-22
 - creating matrices 3-58
 - creating menus 3-60

custom class 3-66
custom menus 3-62
customizing
 with class objects 8-16-8-18
Cut command in Interface Builder 3-49,
 3-61

D

~/ .dbxrc file A-2, A-3
dbx customization variables
 output_dynamic_type A-2, A-4
 scope_look_aside A-4
dbxInfo.o A-20
Debug Application command in Project
 Builder 2-34
Debugger A-1
 Objective C support A-1
debugging
 an application 1-5
debugging an application A-1-A-20
 functions for
 _cb_prompt A-9
 allbrks A-5
 attach A-8
 collOn A-8
 defbrks A-5
 ff A-9
 hex A-10
 hexdump A-10
 memoff A-5
 memon A-4
 morebrks A-5
 objhelp A-4
 pcounts A-6
 pdesc A-6
 penviron A-9
 pg A-10
 pnsstring A-6
 prstar A-6
 pselfvar A-6
 user default variables for
 NSEnableAutoreleasePool A-13

NSEnableDoubleReleaseCheck
 A-13
NSHideOnDeactivateEnabled
 A-13
NSPauseAtStartup A-13
NSSetPoolThreshold A-13
NSShowAllViews A-13
NSShowAllWindows A-13
NSShowDrawTimes A-13
NSShowEvents A-13
NSShowPS A-13
NSShowWindowInfo A-13
NSShowXEvents A-13
NSTrapIllegalFloatingPointOps
 A-13
 using optimized libraries A-20
delegate 3-110
 making your class a 3-156
– **deleteSelection** (IBEditors) D-20
deleting objects 3-49
design guidelines 3-51
– **destination** (IBConnectors) D-4
– **detachObject:** (IBDocuments) D-12
– **detachObjects:** (IBDocuments) D-12
disconnecting objects 3-130
displaying connections 3-122
– **document** (IBEditors) D-20
Document Layout command in Icon
 Builder 5-19
Document menu
 in Icon Builder 5-18
 in Interface Builder 3-171
– **documentPath:** (IBDocuments) D-12
– **drawObject:** (IBDocuments) D-13
– **drawSelection** (IBSelectionOwners)
 D-23
dynamic binding 8-7-8-8, 8-31-8-34
dynamic typing 8-4

E

Edit application
 command reference 4-24
 command-line options 4-2
 and UNIX 4-21
 windows 4-14

Edit class command in Interface Builder
 3-145

Edit menu
 in Edit 4-26
 in Interface Builder 3-173

- **editedObject** (IBEditors) D-20
- **editor:didCloseForObject:**
 (IBDocuments) D-13
- **editorClassName** (NSObject
 Additions) C-12
- **editorForObject:create:**(IBDocuments)
 D-13

Emacs
 commands in Edit 4-21

enabling messageSendDebug A-14

@end directive 8-21

- **establishConnection** (IBConnectors)
 D-4

examining connections 3-122

Expand All command in Edit 4-31

Expand Sel command in Edit 4-31

Expansion Dictionary command in Edit
 4-34

F

fields
 tabbing between 3-128

File's Owner 3-18

file's owner object 3-18

Files menu in Project Builder 2-34

filter match
 setting a breakpoint on A-17

filtering mechanism
 implementing A-19

filters

 adding A-14, A-16
 disabling A-17
 listing current A-19
 removing A-17

Find menu
 in Edit 4-27

- **finishInstantiate** (IBPalette) C-9

First Responder object in Interface Builder
 3-20

Font command in Interface Builder 3-174

Font menu
 in Edit 4-28

fonts 3-84

form field 3-119

Format menu
 in Edit 4-28
 in Icon Builder 5-19
 in Interface Builder 3-173

Foundation Kit 3-141

funcs command in Debugger A-2

G

generating code 3-153

group attributes 3-85

Group command in Interface Builder 3-55,
 3-175

Group in ScrollView command in
 Interface Builder 3-57, 3-175

Group in SplitView command in Interface
 Builder 3-57, 3-175

Group submenu in Interface Builder 3-56,
 3-175

grouping objects 3-34, 3-55, 3-86

H

header file 3-153, 3-162, 3-163, 3-164

Help Builder command in Interface
 Builder 3-134, 3-178

Help Builder panel in Interface Builder
 3-132

Help menu in Edit 4-30
 help, providing in applications 3-132
 – **helpInspectorClassName** (NSObject Additions) C-13
 Hide Grid command in Interface Builder 3-176
 Hide Links command in Edit 4-26
 Hide Markers command in Edit 4-30
 Hide Ruler command in Edit 4-29

I

IB protocol
 specification D-1
 IB_BottomLeftKnobPosition constant E-1
 IB_BottomRightKnobPosition constant E-1
 IB_MiddleBottomKnobPosition constant E-1
 IB_MiddleLeftKnobPosition constant E-1
 IB_MiddleRightKnobPosition constant E-1
 IB_MiddleTopKnobPosition constant E-1
 IB_TopLeftKnobPosition constant E-1
 IB_TopRightKnobPosition constant E-1
 IBAttributesChangedNotification global E-3
 IBCellPboardType global E-3
 IBConnectors protocol
 specification D-3
 IBDidAddConnectorNotification global E-2
 IBDidOpenDocumentNotification global E-2
 IBDidRemoveConnectorNotification global E-2
 IBDidSaveDocumentNotification global E-2
 IBDocuments protocol
 specification D-6
 IBEditions protocol
 specification D-16
 IBInspector class
 specification C-1
 IBInspectors protocol
 specification D-22
 IBMenuCellPboardType global E-3
 IBMenuPboardType global E-3
 IBObjPboardType global E-3
 IBPalette class
 specification C-6
 IBSelectionChangedNotification global E-3
 IBSelectionOwners protocol
 specification D-23
 IBViewPboardType global E-3
 IBWillAddConnectorNotification global E-2
 IBWillCloseDocumentNotification global E-2
 IBWillRemoveConnectorNotification global E-2
 IBWillSaveDocumentNotification global E-2
 IBWindowPboardType global E-3
 icon
 editing 5-3
 Icon Builder application
 command reference 5-18
 icon mode 3-11
 icon position 3-78
id data type 3-163, 3-166, 8-3–8-4, 8-13
 identifying objects
 outline mode 3-127
 with tags 3-107
 identifying outlets and actions 3-143
 IDL template object
 adding to your interface 3-164
 connecting 3-165
 instantiating 3-165
 name 3-165
 IDL type file
 parsing 3-164
 image files in Project Builder 2-16

-
- **imageForViewer** (NSObject Additions) C-13
 - **imageName:** (IBPalette) C-8
 - images
 - adding to project 3-77, 3-80
 - as decoration 3-78
 - associating with buttons 3-75
 - inspecting 3-80
 - @implementation** directive 8-25
 - implementation file 3-154, 3-162
 - implementing a class
 - NSView 3-158-3-161
 - subclass of NSObject 3-155
 - #import** directive 8-23
 - indentation in Edit 4-9
 - inheritance 8-9-8-12
 - **init** method 8-16
 - + **initialize** method 8-19
 - initializing
 - classes 8-19
 - instances 8-16
 - initializing text 3-38
 - **initWithObject:inDocument:** (IBEditors) D-20
 - Insert Field command in Edit 4-34
 - Insert Link command in Edit 4-30
 - Insert Marker command in Edit 4-30
 - inserting custom class 3-162
 - Inspector command in Icon Builder 5-20
 - Inspector command in Interface Builder 3-22, 3-37, 3-80
 - Inspector panel in Interface Builder 3-4, 3-21, 3-22
 - Attributes display 3-63
 - Connections display 3-114, 3-120, 3-122, 3-123, 3-124
 - Help display 3-136
 - Size display 3-36, 3-37, 3-40, 3-98
 - **inspectorClassName** (NSObject Additions) C-13
 - instance
 - generating 3-147
 - instance methods 8-15, 8-22
 - instance variables 8-2-8-3
 - declaration of 8-22
 - inheriting 8-11
 - referring to 8-26-8-31
 - scope of 8-28-8-31
 - Instances display 3-65
 - and attributes 3-65
 - icon mode 3-11
 - making connections 3-115
 - outline mode 3-12
 - instances of a class 8-8
 - allocating 8-16
 - initializing 8-16
 - Instantiate command in Interface Builder 3-147
 - instantiation 3-115
 - NSObject subclass 3-161
 - NSView object 3-162
 - Interface Builder application
 - Alignment panel 3-46
 - application, building 3-4
 - application, testing 3-136
 - attributes, setting 3-63
 - command reference 3-171
 - connections, setting 3-113
 - coordinate system 3-50
 - custom palette, adding 3-169
 - file's owner object 3-18
 - First Responder object 3-20
 - help attachments, reviewing 3-135
 - help, attaching 3-132
 - Inspector panel 3-4, 3-21
 - nib file 3-24
 - nib file window 3-4
 - NSView objects, setting size and position of 3-40
 - object, attaching help 3-132
 - object, examining 3-65
 - overview 3-3
 - Palettes window 3-4
 - preferences, setting 3-167
 - role in class creation 3-138
 - interface design 3-51

@interface directive 8-21
interface files in Project Builder 2-16
interface window 3-9
introspection 8-4, 8-14
isa instance variable 8-4
– **isKindOfClass:** method 8-14
– **isMemberOf:** method 8-14
– **isTestingInterface** (IB) D-2

K

keyboard events 3-20

L

– **label** (IBConnectors) D-4
layering objects 3-57
leaf node 3-87
Line Range command in Edit 4-27
line, making in interface 3-87
Link Inspector command in Edit 4-26
Link Menu in Edit 4-26
Load Tool command in Icon Builder 5-20
localization 2-2

M

main menu 3-60
Make ASCII command in Edit 4-29
Make Column command in Interface Builder 3-48, 3-177
Make Global command in Project Builder 2-34
Make Localizable command in Project Builder 2-35
make program 1-4, 2-23
Make Rich Text command in Edit 4-29
Make Row command in Interface Builder 3-48, 3-177
Make Template Object command in Interface Builder 3-165

makefile 1-4
 preamble and postamble 2-24
– **makeSelectionVisible:** (IBEditors) D-21
managing sounds and images 3-78
Manual command in Edit 4-33
Match command in Edit 4-33
matrix 3-92
 attributes of 3-95
 class (NSMatrix) 3-92
 creating 3-58
 initializing text 3-39
 options 3-98
 selection mode 3-95
– **mazimumSizeForCellSize** (NSCell Additions) C-14
– **maximumSizeFromKnobPosition:** (NSView Additions) C-16
menu cells 3-60
 activating 3-60
 attributes 3-63
 deleting 3-61
 re-sequencing 3-61
Menu palette 3-60, 3-62
menus
 creating 3-60
 custom 3-62
Menus palette 3-28
message expressions 8-5
message receiver 8-5
messages 8-5–8-6
messageSendDebug
 enabling using dwrite A-14
 invoking from a program or Debugger A-15
messaging 8-7–8-8, 8-31–8-34
method 3-91, 3-111, 3-112, 3-151
 for NSButton and NSButtonCell 3-73
methods 8-2–8-3
 class methods 8-15, 8-22
 declaration of 8-22–8-23
 implementation of 8-26
 inheriting 8-11–8-12

methods (*Continued*)
instance methods 8-15, 8-22
of the root class 8-19
overriding 8-11–8-12
return and argument types 8-35
miniaturize button 3-69
– **minimumSizeForCellSize** (NSCell Additions) C-15
– **minimumSizeFromKnobPosition** (NSView Additions) C-16
mouse
actions 3-52
mouse events 3-20
moving objects
to other windows 3-43
within same window 3-33

N

– **nameForObject:** (IBDocuments) D-13
New Application command in Interface Builder 3-23, 3-171
New Attention Panel command in Interface Builder 3-172
New command in Project Builder 2-33
New Empty command in Interface Builder 3-23, 3-172
New Info Panel command in Interface Builder 3-172
New Inspector command in Interface Builder 3-172
New Layout command in Icon Builder 5-19
New Module command in Interface Builder 3-171
New Palette command in Interface Builder 3-172
New Subproject command in Project Builder 2-33
Next Field command in Edit 4-34
nib file
adding classes 3-162
contents 3-24

creating 3-22
main 3-24
owner 3-18
run-time behavior 3-27
saving 3-23
updating 3-163
nib file window in Interface Builder 3-4
Classes display 3-14, 3-142
File's Owner icon 3-19
First Responder icon 3-20
Images display 3-15
Instances display 3-11, 3-12, 3-65, 3-123, 3-124
Sounds display 3-16
– **nibInstantiate** (IBConnectors) D-5
nil 8-3, 8-6
nonretained window 3-68
NSActionCell 3-112
NSApplication Additions specification C-9
NSBox object 3-56
NSBrowser 3-87
attributes 3-87
structure 3-93
NSButtonCell 3-92
NSCell 3-92
NSCellAdditions category specification C-14
NSClipView 3-93
NSControl object 3-19, 3-20, 3-91, 3-111, 3-116, 3-145, 3-151
NSCustomView 3-158
NSForm 3-92
NSObject 3-158
implementing a subclass of 3-155
NSObjectAdditions category specification C-11
NSPanel 3-69
NSPopUpButton 3-94
NSScroller 3-93
NSScrollView 3-81, 3-82, 3-84, 3-93
NSSliderCell 3-92

NSTextField 3-81, 3-84
NSTextFieldCell 3-92
NSTitle 3-81
NSView 3-170
NSView Additions category
 specification C-15
NSView class
 implementing a subclass of 3-158–
 3-161
NSView object
 coordinate system 3-50
 instantiation 3-160
NSWindow 3-69, 3-170

O

Obese Bits command in Icon Builder 5-20
objc_lookupClass() function 8-20
objc_messageMatchedFilter() A-15, A-16
objc_messageSendDebug A-13
objc_msgSend() function 8-31
Object class 8-9, 8-10
object pointer A-2
 dynamic type A-2
 static type A-2
– **object:** (IBInspector) C-3
objects 8-2–8-3
 aligning 3-47
 arranging 3-45
 automatic resizing of 3-98
 compound 3-91
 connecting 3-113
 deleting from nib file 3-49
 disconnecting 3-130
 group attributes 3-86
 grouping 3-55, 3-86
 identifying 3-127
 identifying with tags 3-107
 layering 3-57
 making columns and rows 3-48
 making same size 3-52
 matrix of 3-58
 minimum size 3-54
 moving 3-33, 3-43
 multiple selection 3-34
 placing 3-33
 positioning 3-33, 3-40
 removing 3-49
 selecting multiple 3-34
 setting titles 3-38
 shrinking 3-54
 sizing 3-39, 3-52, 3-54
 tracing when debugging A-13
 ungrouping 3-56
– **objects:** (IBDocuments) D-13
– **ok:** (IBInspector) C-3
– **okButton:** (IBInspector) C-4
Open command
 in Interface Builder 3-7
 in Project Builder 2-33
Open Folder command in Edit 4-25
Open in Workspace command in Project
 Builder 2-34
Open Makefile command in Project
 Builder 2-33
Open Selection command in Edit 4-25
– **openEditorForObject:** (IBDocuments)
 D-14
openfile shell command 4-13
– **openSubeditorForObject:** (IBEditors)
 D-21
– **orderFront** (IBEditors) D-21
– **originalWindow** (IBPalette) C-9
outlet 3-5, 3-109
 adding to class 3-144
 and outline mode 3-125
 autotyping 3-166
 connecting 3-114
 naming 3-144
 specifying 3-143
outline mode 3-12
 and connections 3-124
 disconnecting objects 3-131
 identifying objects 3-127
 making connections 3-120
overriding methods 8-11–8-12

P

- Page Layout command
 - in Edit 4-28
 - in Interface Builder 3-174
- palette
 - Menus 3-28
 - objects 3-27
 - TextViews 3-30
 - using 3-27
 - Views 3-28
 - Windows 3-30
- palette project in Project Builder 2-2
- **paletteDocument** (IBPalette) C-9
- Palettes command 3-27
- Palettes submenu in Interface Builder 3-27, 3-178
- Palettes window in Interface Builder 3-4
- panel
 - attributes 3-67
 - customizing 3-67
 - difference from window 3-70
 - sizing 3-36
- **parentOfObject:** (IBDocuments) D-14
- Parse command in Interface Builder 3-163
- Parse IDL command in Interface Builder 3-164
- Paste and Link command in Edit 4-26
- Paste command in Interface Builder 3-44, 3-49
- Paste Ruler command in Edit 4-30
- **pasteInSelection** (IBEditors) D-21
- **pasteType:fromPasteboard:parent:** (IBDocuments) D-14
- path** environment variable 1-6
- **perform:** method 8-36
- **perform:with:** method 8-36
- **perform:with:with:** method 8-36
- Pipe command in Edit 4-22, 4-32
- pixels inset 3-73
- **placeView:** (NSView Additions) C-17
- placing objects 3-33
- polymorphism 8-6
- pop-up list 3-90
 - attributes 3-89
 - structure 3-94
- positioning objects 3-33
 - precisely 3-40
- postamble file 2-24
- preamble file 2-24
- Preferences command in Edit 4-3
 - C options 4-11
 - global options 4-6
 - text options 4-8
 - user options 4-4
- Preferences command in Header Viewer 6-14
- Print command in Interface Builder 3-62
- printing 3-119
- @private** directive 8-29–8-31
- project 2-1
 - building 2-18
 - converting earlier version 2-6
 - creating 2-3
 - debugging 2-32
 - directory 2-15
 - files, managing 2-14
 - makefile 1-4
 - opening 2-5
 - running 2-32
- project attributes 2-8
 - application 2-9
 - bundle 2-11
 - palette 2-13
 - subproject 2-11
- Project Builder application 3-2, 3-6, 3-7, 3-134, 3-155, 3-162
 - build targets 2-23
 - command reference 2-33
 - preferences, setting 2-29
 - project attributes, setting 2-8
 - project files, managing 2-14
- Project menu in Project Builder 2-33

project window in Project Builder 2-5
 Attributes display 2-8
 Builder display 2-18
 Files display 2-14
@protected directive 8-29–8-31
@public directive 8-29–8-31
pull-down list 3-89, 3-90

Q

quitting test mode 3-136

R

radio button 3-71, 3-96
receiver of a message 8-5
reference object 3-41, 3-48, 3-52
Remove command in Project Builder 2-34
– **removeConnector:** (IBDocuments) D-14
removing objects 3-49
– **replaceObject:withObject:**
 (IBConnectors) D-6, D-15
– **resetObject:** (IBEditors) D-22
– **resignKeySelectionForEditor:** (IBEditors)
 D-15
resize bar 3-69
resizing
 automatically 3-98
 of objects 3-39, 3-40
resources 3-75
– **respondsTo:** method 8-38
retained window 3-69
– **revert:** (IBInspector) C-4
– **revertButton:** (IBInspector) C-5
Rich Text Format in Edit 4-8, 4-29
root class 3-137, 3-155
rows of objects 3-48
RTF See Rich Text Format in Edit
Run Application command in Project
 Builder 2-34

S

Same Size command in Interface Builder
 3-53, 3-177
Save All command in Edit 4-25
Save As command in Edit 4-25
Save command in Edit 4-25
Save To command in Edit 4-25
saving nib file 3-23
scroll view 3-81
 structure 3-93
SEL data type 8-34
sel_getName() function 8-35
sel_getUid() function 8-35
Select All command in Interface Builder
 3-34
Select in Workspace command in Project
 Builder 2-34
selecting multiple objects 3-34
selection mode (matrix) 3-95
– **selection:** (IBSelectionOwners) D-23
– **selectionCount** (IBSelectionOwners)
 D-24
– **selectionOwner** (IB) D-2
– **selectObjects:** (IBEditors) D-22
@selector() directive 8-34
selectors 8-7, 8-34–8-35
self 8-39–8-42, 8-43–8-45
Send To Back command in Interface
 Builder 3-57
Set Grid Off command in Interface Builder
 3-176
Set Grid On command in Interface Builder
 3-176
Set Name command in Interface Builder
 3-173
– **setName:forObject:** (IBDocuments)
 D-15
– **setSelectionFromEditor:**
 (IBDocuments) D-15

setting breakpoints in Objective C
 methods A-2

setting the font 3-84

setting titles of objects 3-38

Show Grid command in Interface Builder
 3-46, 3-176

Show Links command in Edit 4-26

Show Markers command in Edit 4-30

Size display 3-36, 3-37, 3-40, 3-98

Size submenu in Interface Builder 3-177

Size to Fit command in Interface Builder
 3-54, 3-55, 3-177

– **sizeInspectorClassName** (NSObject
 Additions) C-13

sizing objects 3-39, 3-52, 3-53
 precisely 3-40

sizing windows and panels 3-36

Sort command in Project Builder 2-34

sound files in Project Builder 2-16

sounds
 adding to project 3-77, 3-80
 associating with buttons 3-75
 inspecting 3-80

Source command in Edit 4-33

– **source** (IBConnectors) D-6

Spelling command in Edit 4-26

static typing 8-13, 8-20

Structure menu in Edit 4-15

Subclass command in Interface Builder
 3-141, 3-142

subclasses 8-9
 creating 3-141

subclassing 3-141

subproject, project in Project Builder 2-2

super 8-39–8-43

superclass 8-9
 selecting 3-141

switch button 3-71, 3-96

T

tabbing between fields 3-128

tags 3-107

tags file 4-23, 4-33

target 3-91, 3-111, 3-116, 8-37

target-action paradigm 8-36–8-37

Test Interface command in Interface
 Builder 3-136, 3-172

test mode 3-101, 3-137

testing an interface 3-136

text
 alignment 3-84
 font of 3-84
 initializing 3-38
 setting attributes 3-84

Text command in Interface Builder 3-174

text field 3-81

Text menu in Edit 4-29

– **textDidBeginEditing:** (IBInspector) C-5

TextViews palette 3-30

titles 3-38

Tools menu in Interface Builder 3-177

top-level object 3-4

– **touch:** (IBInspector) C-5

– **touch** (IBDocuments) D-16

U

UI design 3-51

Undelete command in Edit 4-26

Ungroup command in Interface Builder
 3-56, 3-175

UNIX
 displaying manual pages in Edit 4-33
 piping output into Edit 4-22
 using a tags file in Edit 4-23
 utility commands in Edit 4-31

Unnest command in Edit 4-29

Unparse command in Interface Builder
 3-154

Update Directory command in Edit 4-34

updating class definition 3-163
User Commands menu in Edit 4-22, 4-32
User Pipes menu in Edit 4-22, 4-32
using palette 3-27
using tags 3-107
Utilities menu in Edit 4-31

V

– **validateEditing** (IBEditors) D-22
Views palette 3-28

W

– **wantsButtons** (IBInspector) C-5
– **wantsSelection** (IBEditors) D-22
window
 attributes 3-67
 backing (buffer) 3-68
 buffered 3-69
 controls 3-69
 coordinate system 3-50
 customizing 3-67
 difference from panel 3-70
 nonretained 3-68
 options 3-70
 resize bar 3-36
 retained 3-69
 setting dimensions 3-36
 sizing 3-36
– **window** (IBEditors) D-22
– **window** (IBInspector) C-5
Windows palette 3-30