Interface Builder™

and

Object-Oriented Design

in the

NeXTstep® Environment

A SCaNeWS Supplement

by

Michael K. Mahoney

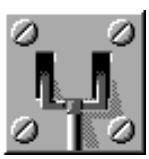
California State University, Long Beach

mahoney@csulb.edu

The original version of this tutorial was presented on 29 April 1991 at CHI '91 in New Orleans.
CHI is the Computer-Human Interface Conference sponsored by ACM-SIGCHI.

Copyright ©1991 by Michael K. Mahoney

Portions of this tutorial taken from the NeXT manuals are
copyright ©1990 NeXT Computer, Inc.
and are used herein with permission.



Abstract

NeXTstep's Interface Builder and Application Kit objects are powerful tools which drastically reduce the cost of developing applications with graphical user interfaces. By working through this tutorial you will learn how to build NeXTstep 2.0 user interfaces (mostly in a graphical fashion), make connections between user interface (and other) objects and generate skeletal Objective-C language files without any programming! You'll also get an introduction to object-oriented programming using the Objective-C language. The only prerequisite is a knowledge of the C programming language.

The tutorial begins with a brief overview of the NeXTstep user interface and the objects in its Application Kit. Then step-by-step instructions for building several user interfaces for various applications are given. By performing these steps you'll learn how to drag Application Kit objects (e.g. windows, buttons, forms, menus) from interface object palettes into a developing application's windows and then customize these objects to suit an application. You'll set up interactions between these and custom objects (where the computational engine of an application resides) by drawing connection lines. Most of the types of operations available in Interface Builder will be used in these applications.

Projects 1 and 2 are mainly concerned with showing how to use Interface Builder to build simple interfaces and contain almost no programming at all. Project 3 contains an introduction to displaying custom graphics. Project 4 contains an introduction to handling mouse events. Project 5 shows how to read information from a class interface file into Interface Builder. Project 6 shows how to set up your own interface object as an integral part of Interface Builder. The content of several of Interface Builder's windows will be discussed after most projects. Other tools such as an icon builder and sound editor will be used to build user interface features.

Instructor Biography

Michael K. Mahoney is a Professor and chair of the Computer Engineering and Computer Science at California State University, Long Beach. He started programming a NeXT Computer in January 1989, attended a developer workshop given by NeXT in May 1989 and conducted three semesters of NeXT software development seminars at CSULB. He has given presentations on Interface Builder at ACM meetings in Seattle and Los Angeles and presented the original version of this tutorial at CHI'91 in New Orleans. He is currently directing several students who are developing 3D computer graphics applications in the NeXTstep environment.

Mike is founder and President of SCaN, the Southern California NeXT Users' group, which meets monthly at various sites in Los Angeles County. He is also co-editor of *SCaNeWS*, which together with this tutorial can be obtained from the nova.cc.purdue.edu archive site and possibly others (see page 75).

Mike earned his Ph.D. in mathematics at the University of California, Santa Barbara in 1979. He has published papers in computer graphics, computer science education and mathematics. He won campus-wide teaching awards at both UCSB and CSULB.

5/7/91

Michael K. Mahoney

Computer Engineering and Computer Science Dept
California State University, Long Beach
Long Beach, CA 90840

(213) 985-1550
e-mail: mahoney@csulb.edu or
mahoney@beach.csulb.edu

Table of Contents

		Page
Abstract		2
Instructor Biography		3
Table of Contents		4
NextApps and NextDeveloper Apps for Application Development		6
NeXTstep User Interface (Window Types)		7
NeXTstep User Interface (WriteNow Examples)		8
NeXTstep Menu Structure (WriteNow Examples)		9
NeXTstep Panels (WriteNow Examples)		10
Building an Icon with the Icon Application		12
Apps & Interface Builder (IB), Classes & Objects		13
Application Kit Hierarchy of Classes		14
Project 1A - OneButton (which makes a Sound)		15
Project 1A Step by Step		16
New Application in IB	(screen dump)	18
OneButton in IB	(screen dump)	19
OneButton in Workspace	(screen dump)	20
Project 1A Wrap-up		21
The **Files Created by IB**		22
The **File Window**		23
The **Sounds Window**		24
Project 1B - OneButton (revised)		25
Project 1B Step by Step		26
OneButton with Icons in IB	(screen dump)	30
OneButton with Application Icon in WM	(screen dump)	31
Project 1B Wrap-up		32
The **Icons Window**		33
The **Palettes Window** (s)		34

(continues)

Table of Contents (continued)

		Page
The Inspector Window(s)		36
The Project Inspector		37
Project 2 - Convert Kilometers to Miles (KmToMi)		38
The 4 Messages in KmToMi		39
Project 2 Step by Step		40
Outlet Connection	(screen dump)	44
Unparse and performClick	(screen dump)	45
KmToMi in Workspace	(screen dump)	46
Project 2 Wrap-up		47
The Classes Window		48
Operations on Classes		49
The Application Development Process		50
Project 3 - SimpleGraphics (custom drawing in a window)		51
Project 3 Step by Step		52
SimpleGraphics in IB	(screen dump)	55
SimpleGraphics in Workspace	(screen dump)	56
SimpleGraphics Objective-C Files		57
Project 3 Wrap-up		61
Project 4 - PenDraw (handling Mouse Events)		62
Project 4 Step by Step		63
PenDraw Objective-C Language Files		65
Project 5 - Parse Example (making a class known to IB)		70
Project 5 Step by Step		71
Project 6 - SketchPalette (customizing Interface Builder)		72
What to do next?		73
Brief Glossary (mainly Objective-C terms)		74
References		75
Acknowledgments and Trademarks		76

NextApps and NextDeveloper Apps for Application Development

The following application programs (which reside in the **/NextApps** or **NextDeveloper/Demos folders)** are useful for application development.

Interface Builder

Digital Librarian™ (can search for Application Kit objects and methods, etc. in NeXT manuals)

Edit (editor for writing code)

Terminal (UNIX shell)

Yap (for writing and testing PostScript® code)

Icon (for creating custom icons; a "CHI_Icon" will be created and used later in IB projects)

BusyBox (for learning about the NeXTstep User Interface), in the **/NextDeveloper/Examples** folder

Other useful applications for application development may be found in the **/NextDeveloper/Apps** folder.

NeXTstep User Interface Window Types

- **Standard Window -** the main working area of an application (e.g. the window containing the file being edited in a word processor)
- **Panel -** used to give instructions to an application
- **Menu -** contains a list of commands
- **Pop-up list -** menu-like list that appears on top of a button when the button is pressed
- **Pull-down list -** menu-like list that appears below a button when the button is pressed
- **Miniwindow -** small titled window representing a window that's been miniaturized
- **Freestanding (or docked) icon -** represents the whole application

The key window is the window associated with keyboard actions**. The main window is** the standard window where a user is currently working (and is usually also the key window).

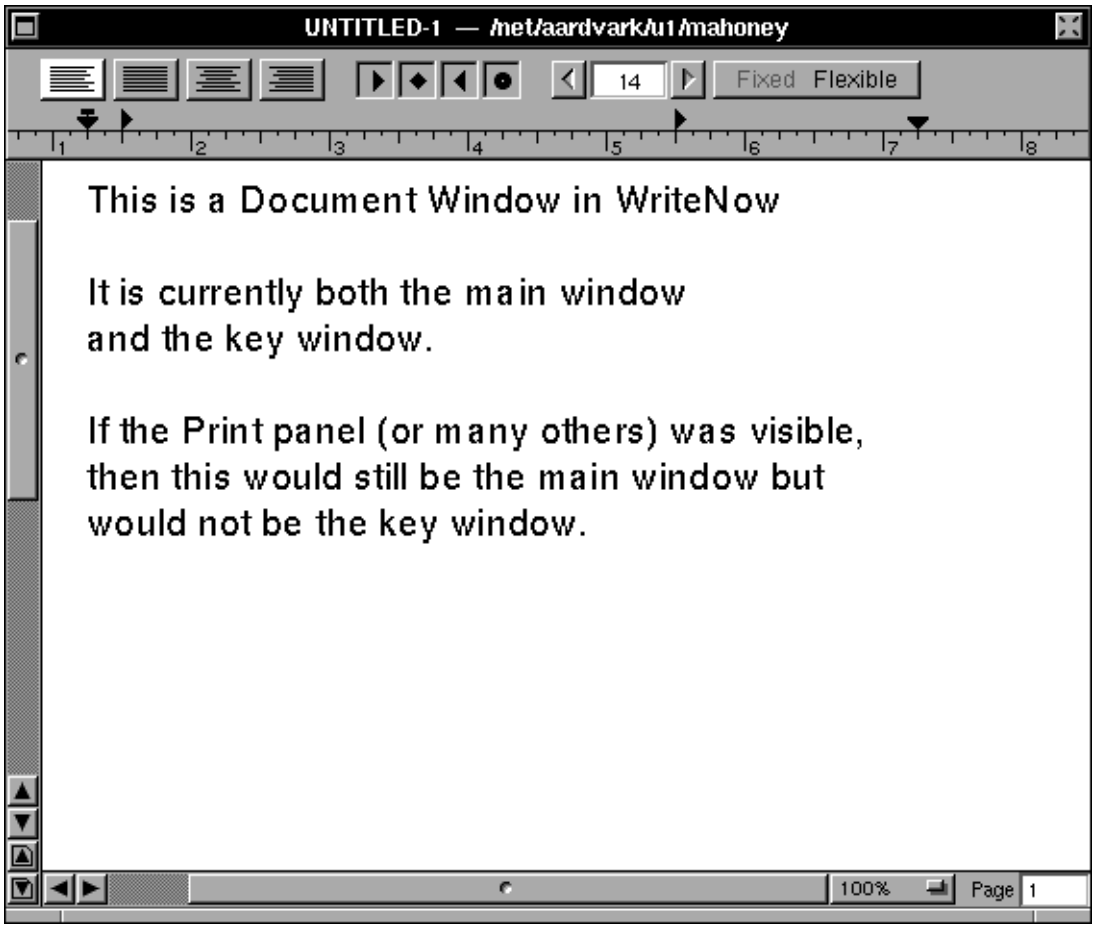
NeXTstep User Interface

(examples from the WriteNow® wordprocessor)

Application Icon
(Dock, File Viewer,
or freestanding)

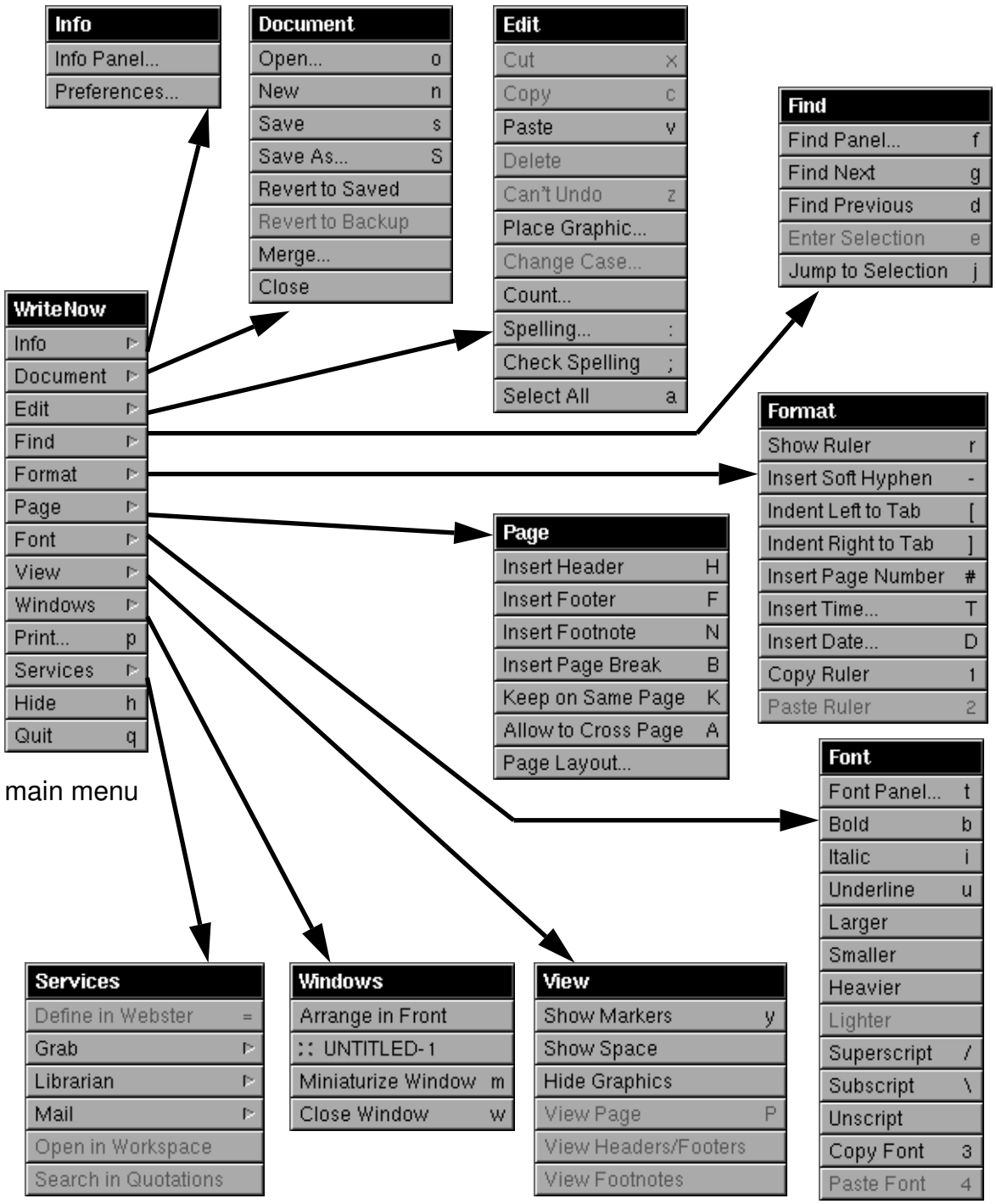
Document Icon
(File Viewer)

Miniwindow
(freestanding)



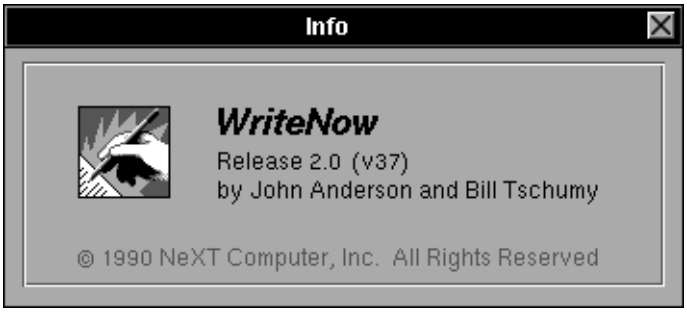
NeXTstep Menu Structure

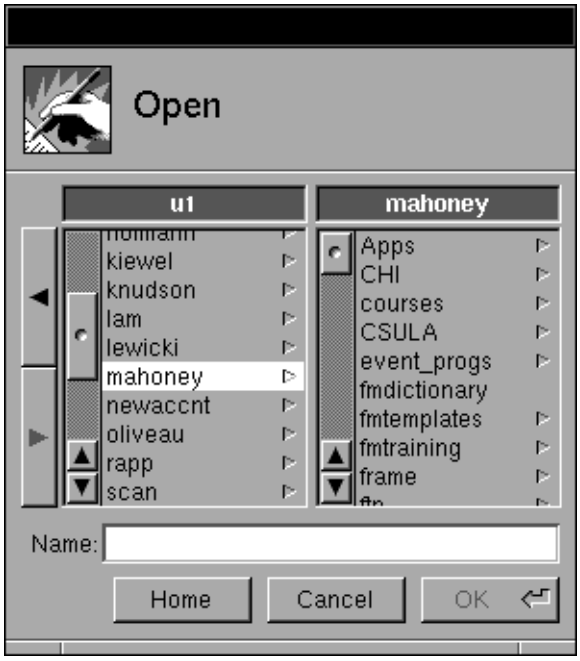
(examples from WriteNow)

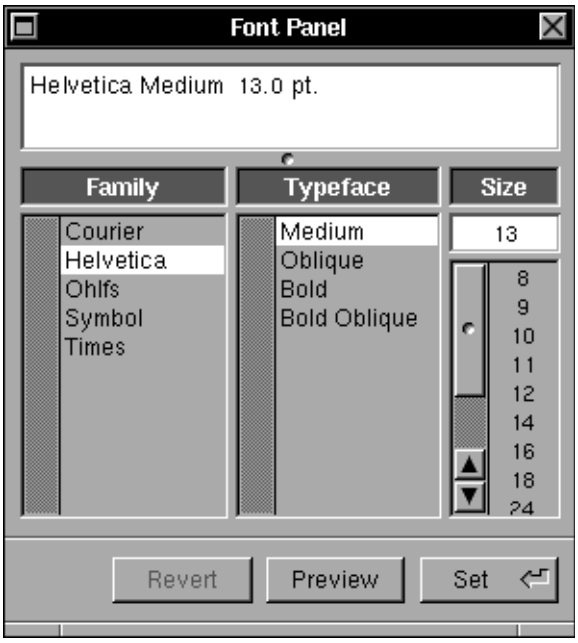


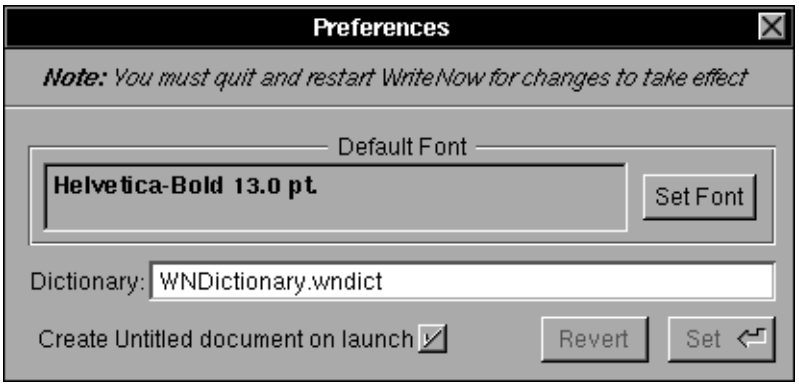
NeXTstep Panels

(examples from WriteNow)



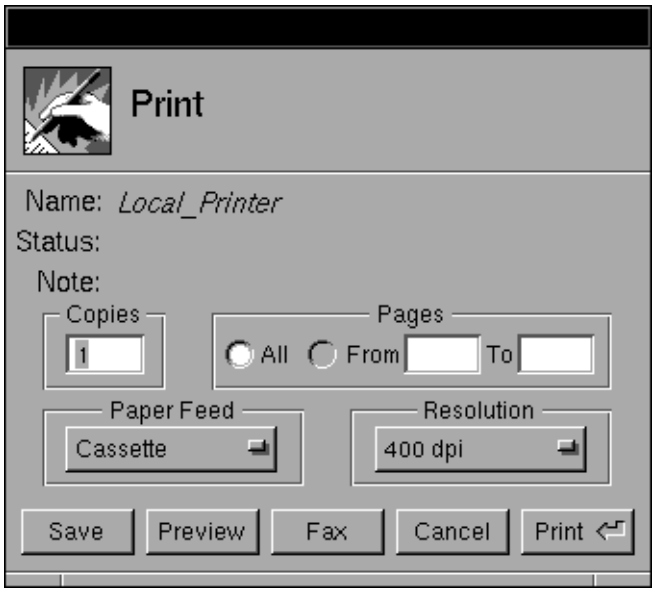


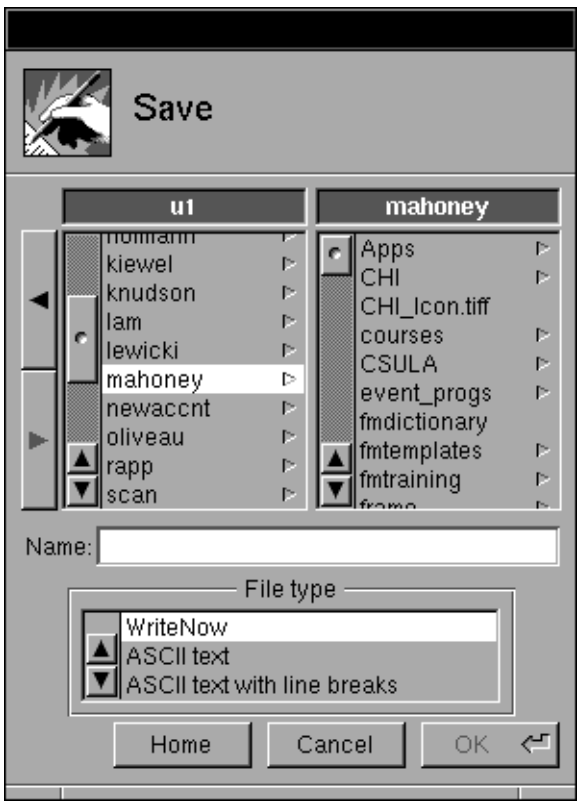


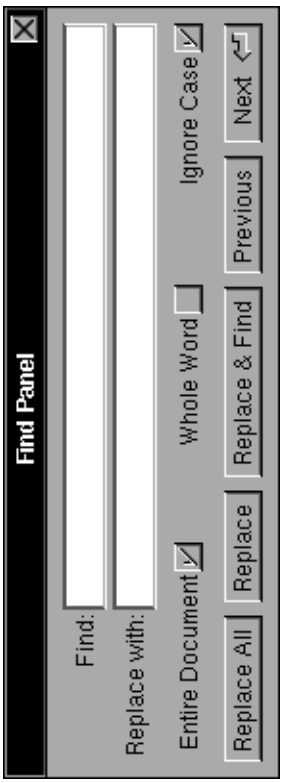


NeXTstep Panels (cont.)

(examples from WriteNow)







Building an Icon with the Icon App

By going through the steps below we will build an application icon, one which can be used to identify an application in the Workspace.

When we're done it will look something like this:

We associate an application icon with an application while in Interface Builder. It must be 48 x 48 pixels in size and have an alpha channel. An icon with these properties can also be used to identify document types in the Workspace.

1. **Launch the Icon application** by double-clicking on its icon (in the /NextDeveloper/Demos folder).

2. **Open an existing icon file** (e.g. BusyBoxApp.tiff in the /NextDeveloper/Examples/BusyBox folder) by choosing **Open** in Icon's **Image** submenu and finding the ".tiff" file. (This will insure that the size and properties will be those of an application icon.)

3. **Choose the rectangle** (next to the white circle) **in the Tools Palette**.

4. **Change the rectangle color to gray** by choosing "Colors" in the Tools submenu to open the Colors panel and then clicking on the color in the panel.

5. **Drag across the BusyBox icon so it's completely gray**.

6. **Choose the pencil** in the **Tools palette**.

7. **Change the color of the pencil to black** (in the Colors panel) and **make the pencil width larger** by clicking in the "**Line Width**" box in the **Inspector panel**.

8. **Draw a "CHI symbol"** or whatever else you want on top of the square gray icon.

9. **Save the image in your home folder** with the name "**CHI_Icon.tiff**" by choosing "**Save As**" in the **Image** submenu and typing "~/**CHI_Icon.tiff**".

10. **Quit the Icon Application**.

11. **Drag the "TIFF" icon** titled "**CHI_Icon.tiff**" onto the File Viewer's **Shelf**.

Applications and Interface Builder

An application can be thought of as having two parts

1. a **user interface part**, and
2. a **computational part** (where the programming logic unique to your application, the non-interface code resides).

Interface Builder is a tool that

- lets you graphically (for the most part) specify your application's user interface part which consists of objects
- sets up the corresponding objects (e.g. buttons, menus, windows) for you
- lets you add your own custom objects (which may contain the computational part)
- makes it easy for you to establish connections between these objects so that messages can be passed between them
- greatly speeds up the application development process.

Classes and Objects

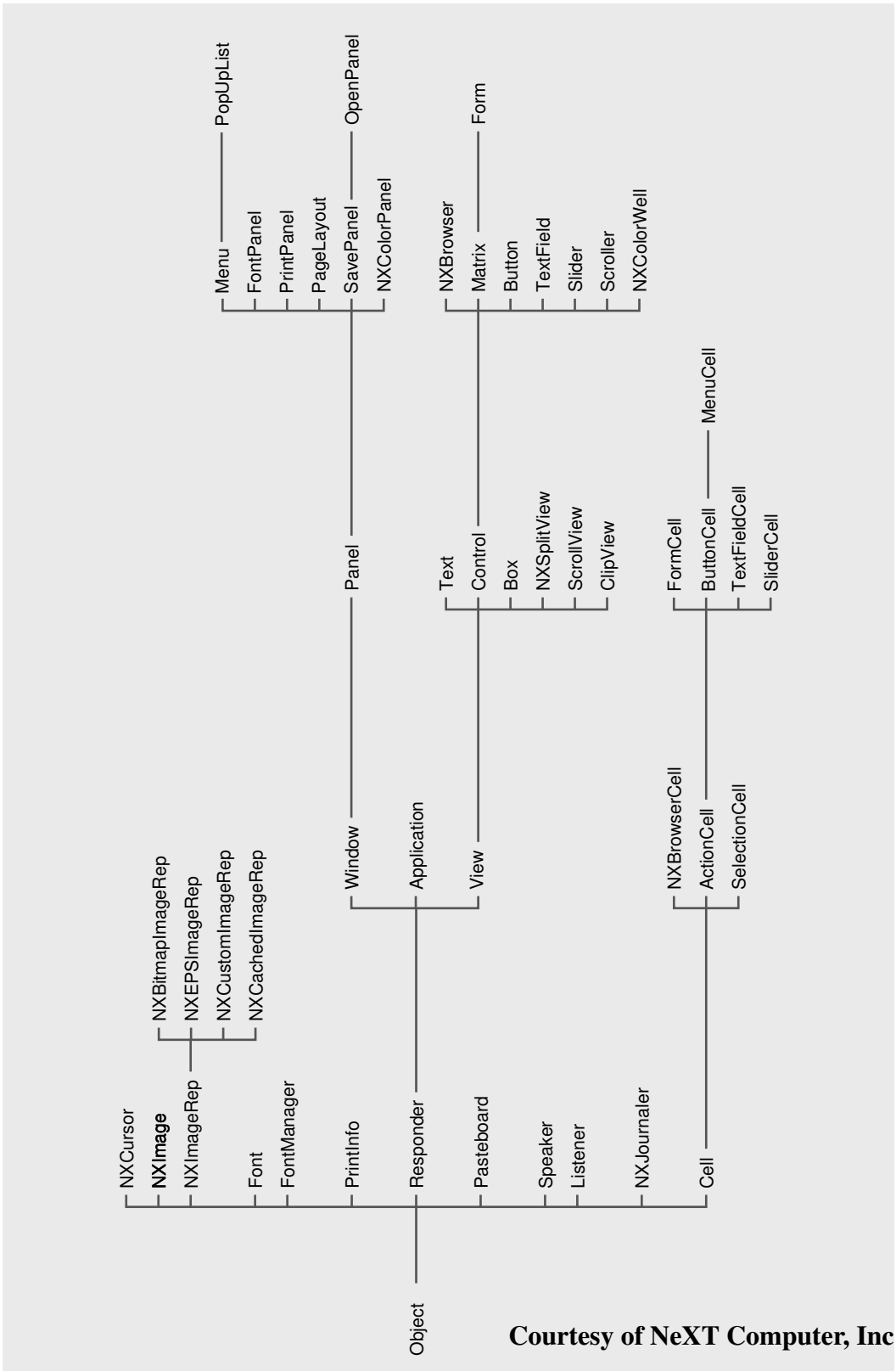
- All NeXTstep applications use the Application Kit to implement a window-based user interface.
- Buttons, sliders, menus, windows and more are defined as classes in the Application Kit.
- Each class inherits both instance variables (data structures) and methods (procedures) from all the classes above it in the hierarchy.

In the projects in this tutorial, we will use the Application Kit to

1. create objects (instances) belonging to the classes defined in the Kit (e.g. we'll create buttons, windows, menus, etc.)
2. define new subclasses of Application Kit classes and create instances of the subclasses for our applications (which may contain new "instance" variables and override methods)

Analogy: The following rough analogy between classes and objects comes from Pascal (or C). A class is like a type declaration while an object is like a variable of that type. Of course, there's much more to classes and objects than types and variables (e.g. inheritance, encapsulation, methods).

The Application Kit Hierarchy of Classes



Courtesy of NeXT Computer, Inc

Project 1A - OneButton

(a very simple application with a window containing a single button which makes a sound when pressed)

When Project 1A is complete, the main menu and window of OneButton will look like

The **objectives** of Project 1A are to learn how to use Interface Builder to

- create a complete application from scratch
- drag a button into an application's interface
- add a sound to a button
- test an application's interface (while in IB)
- "make" an application.

Project 1A Step by Step

1. **Launch Interface Builder (IB)** by double-clicking its icon (which resides in the /NextApps folder). Only the main IB menu and **Palettes window** are visible.

2. Choose **New Application** from the File submenu.

   Three more windows appear - the **File window** containing icons, the **main menu ("UNTITLED")** and **main window ("My Window")** of the new application.

3. Choose **Save As** from the File submenu and enter **"proj1/OneButton"**. Click on **Create** when prompted to create a **new folder called "proj1"**.

   The interface file **"OneButton.nib"** will be saved in this newly created folder.

4. Choose **Project** from the File submenu and click the **OK** button in the **Project Inspector** (which opens when Project is chosen) to create a project of the Application type. (Other types of projects are discussed in Project 6.)

   The four files "**IB.proj", "OneButton.iconheader", "OneButton_main.m**" and **"Makefile**" will be saved in the proj1 folder.

   The content of these files is discussed on page 22. See the **"New Application in IB" screen dump** on page 18. Note the files in the Workspace Manager's (**WM**) **File Viewer**.

5. **Add a button to the application's interface** by dragging a **"Button" icon** from the **Palettes window** into "My Window".

6. **Make the button larger** by dragging on its handles.

7. **Rename the button to "Funky Sound"** by double-clicking on the name "Button" and typing (without a Return) the new name.

8. **Enlarge the "Funky Sound" text** by repeatedly clicking **Larger** in IB's Format submenu.

9. **Make "My Window" smaller** by dragging on the 3 parts of its resize bar (at the window's bottom).

10. **Force the button to make a sound when clicked** by performing steps 11-14 below.

11. **Open the Sounds window** by double-clicking on the Sounds suitcase icon in the File window. Drag on its slider to find the sound icon labeled "**Funk**".

12. **Open the Sound Inspector** by double-clicking the “Funk” icon in the Sounds window. We can inspect or modify sounds in this inspector.

13. **Play the sound “Funk”** (a system sound) by clicking the **Play** button in the Sound Inspector.

14. **Add the “Funk” sound to the “Funky Sound” button** by dragging the “Funk” sound icon from the Sounds window and releasing it on top of the button.

15. **Test OneButton’s interface while in IB** by choosing **Test Interface** from IB’s File submenu. Click on the “Funky Sound” button to hear the “Funk” sound.

16. **End the interface test** by double-clicking on the **big switch icon** (which replaces the IB icon during interface testing) or by choosing Quit from OneButton’s main menu.

17. Choose **Save** from the File submenu to **update the “OneButton.nib” file** in the proj1 folder.

18. Choose **Make** from the File submenu.

    A (UNIX) **shell window** will open (from the Workspace Manager, not the Terminal app) and the application will be **compiled** into an executable file called **“OneButton.debug”** (which contains symbol information for the GDB debugger). The interface we created will be included in this file.

    The Screen should look much like the “**One Button in IB**” screen dump on page 19.

19. **Quit Interface Builder** by choosing **Quit** from the Main menu (not necessary but simplifies the screen).

20. **Highlight the OneButton.debug icon** in the WM’s File Viewer and drag it onto the **Shelf. Do the same for the OneButton.nib icon.**

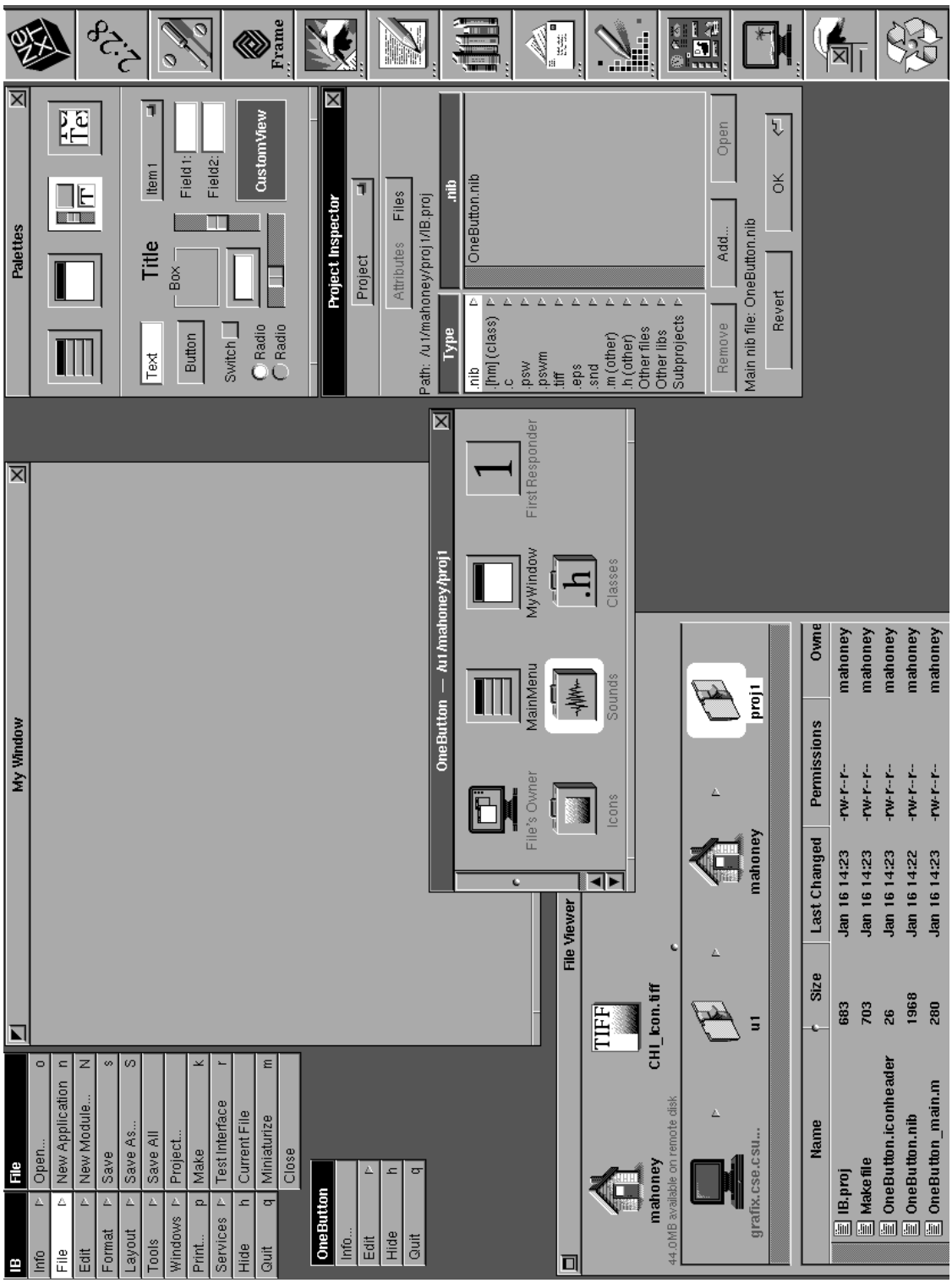
21. **Launch the program OneButton** by double-clicking on the OneButton.debug icon on the Shelf.

    The screen should look much like the **“OneButton in Workspace” screen dump** on page 20. Note the listing of files at the bottom of the File Viewer.

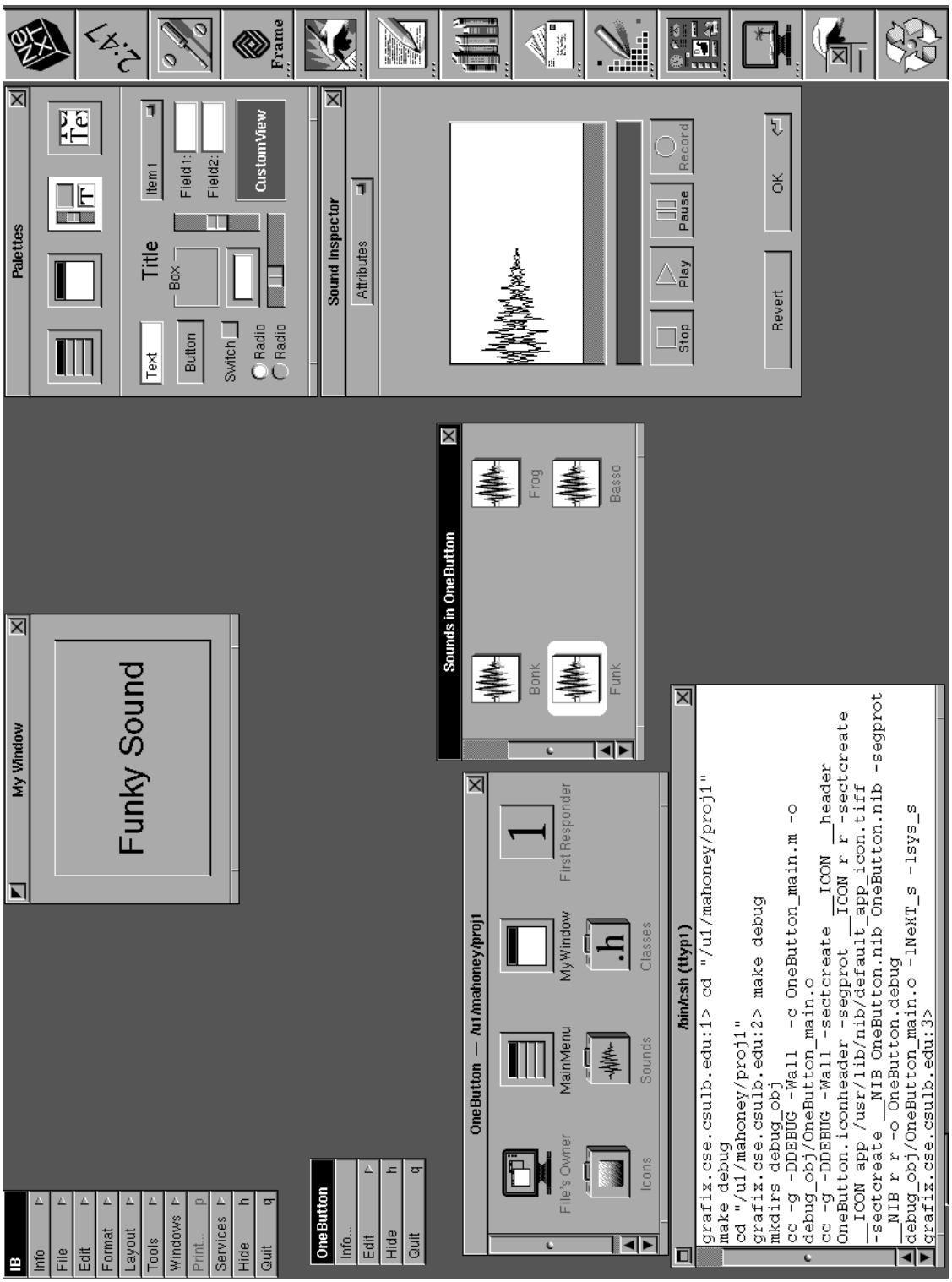
22. **Click the “Funky Sound” button** and listen to the sound. Choose each menu item and see what happens. (Only “Hide” and “Quit” work properly.)

23. **Quit OneButton** by choosing **Quit** from the Main menu. **Close the UNIX shell** as well. Go to page 21 for a wrap-up of Project 1A.

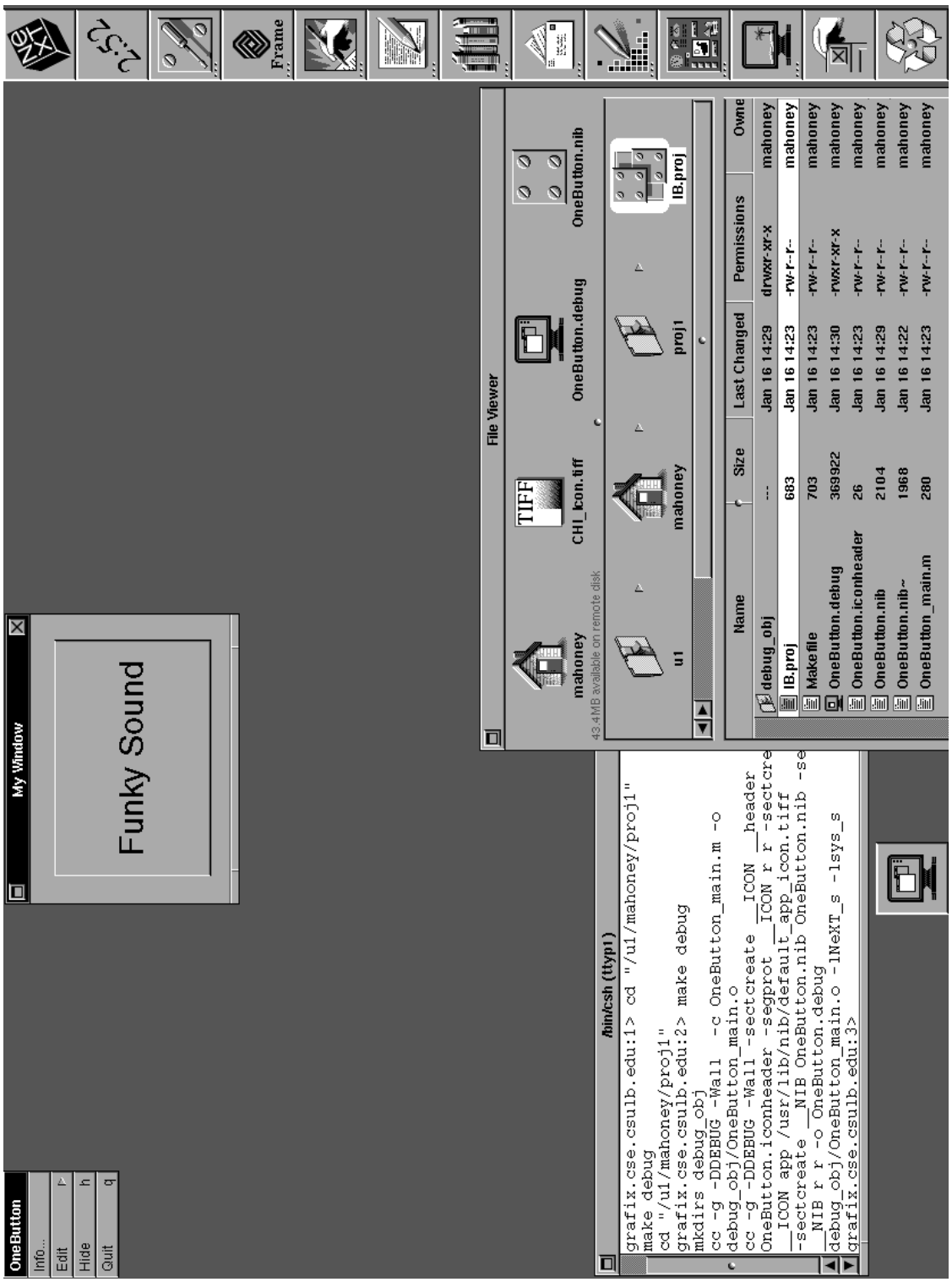
New Application in IB



OneButton in IB



OneButton In Workspace



Project 1A Wrap-up

We

- designed a very simple application consisting entirely of interface objects (a menu and a window containing a button which produces sound when pressed)
- tested the interface while still in Interface Builder
- compiled the system to get an executable file (via a command which invokes the UNIX® “make” utility)
- executed (launched) the application from the Workspace.

Note that Project 1A did not contain a computational engine (e.g. a calculation in response to the clicking of a button) and thus we didn’t need to write any code.

This rarely occurs.

Before revising OneButton (in Project 1B on page 25) we discuss the files created by Interface Builder, and the File and Sounds windows.

The Files Created by IB

In Project 1A Interface Builder created the following 5 files:

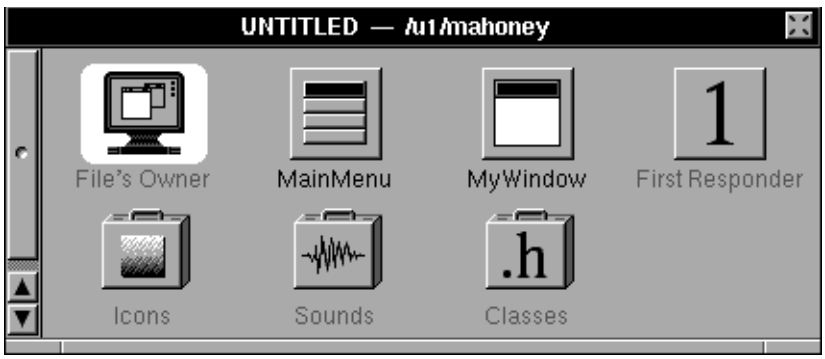
- **OneButton.nib** (which saves the interface specifications made in Interface Builder)
- **IB.proj** (which keeps the parts of the project organized)
- **Makefile** (specifies which files are used by the compiler and linker to build the application; a UNIX “make” file with extra NeXT goodies)
- **OneButton_main.m** (the main program file which contains the main() Objective-C function)
- **OneButton.iconheader** (which saves the information about the application and document icons - see Project 1B)

Interface Builder will also create **Objective-C class interface (.h)** and **implementation (.m) files** for a new class when the **Unparse** command is given.

We’ll see this in Project 2 on page 38.

The File Window

- contains icons which represent the **objects** that make up your application and the **resources** that can be assigned to certain objects in your application.
- Below is the File window for a new application



- **Square icons represent objects**

  These icons can be used as the **source or destination of connections** - we'll see examples of this in Project 2.

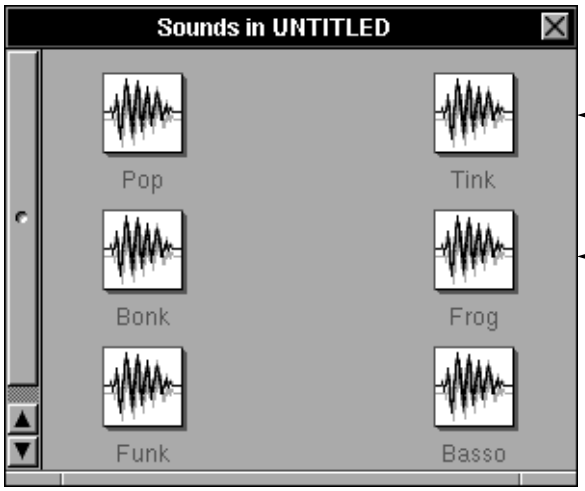
- **Suitcase icons represent resources**

  Available resources are Icons, Sounds and Classes. Double-clicking on a suitcase icon opens a window of resources.

- Other object icons may appear in this window as objects are added to the interface (see Project 1B).

The Sounds Window

- Double-clicking on the suitcase icon labeled “Sounds” in the File window opens the Sounds window below



Sounds readily available for the application’s interface.

These six are system sounds and cannot be edited. They can be copied and the copy can be renamed and edited.

- Icons representing other sounds may be created by another application (e.g. **SoundPlayer** in the /NextDeveloper/Demos folder) and dragged into this window from the Workspace. These sounds will then be available for use in the interface being built in IB.
- To inspect one of these sounds, double-click on its icon and use the **Sound Inspector**.
- To make an interface button produce sound when pressed, drag the sound icon from this window and release it on top of the button.

Project 1B - OneButton (revised)

(add panels, menus, and icons to Project 1A)

The **objectives** of Project 1B are to learn how to use Interface Builder to

- add menu items
- add customized Info and Help panels
- make connections between these menu items and panels in order to force the panels to appear when the menu items are chosen
- add customized text to panels
- add an icon (previously created by the **Icon** application) in several different ways
  1. on a button
  2. in Info and Help panels
  3. as an application icon (which will appear in the Workspace).

For a glimpse of the finished product, see page 31.

Project 1B Step by Step

Before starting Project 1B make sure that the **OneButton.debug**, **OneButton.nib,** and **CHI_Icon.tiff** icons are on the File Viewer's **Shelf**.

1. **Simultaneously launch Interface Builder and bring the OneButton interface into IB** by double-clicking on the "**OneButton.nib**" icon on the Shelf.

2. **Make the CHI_Icon available to the OneButton interface** by dragging the **CHI_Icon.tiff** icon from the File Viewer's Shelf and releasing it on top of the **Icons suitcase** in the IB's File window.

   Add the CHI_Icon to the project and **copy it into the proj1 folder** by clicking on default buttons in the panels which open. The **Icons window** opens. Drag on its slider until the **CHI_Icon appears**.

3. **Open the NXImage Inspector** by double-clicking on the CHI_Icon in the Icons window (not necessary but allows us to inspect the icon). Note that its dimensions are 48 x 48, the correct size for an application icon.

4. **Make a copy of the CHI_Icon appear on the "Make Sound" button** by dragging the CHI_Icon from the Icons window and releasing it on top of the button. Note that the **Button Inspector** opens.

5. **Force the button's title to be above the icon** by clicking in the Button Inspector at the top of the "Icon Position" diamond and then OK.

6. **Rename OneButton's "MyWindow" to "OneButton App"** by clicking in it (but not on the button), clicking in the **Window Inspector** (which opens when "My Window" is clicked) and entering the new name (finish with a **Return or click OK**).

7. **Customize OneButton's menu** by performing steps 8-11 below.

8. **Open the Menus palette** by clicking on the left icon at the top of the Palettes window.

9. **Add an Info submenu to OneButton's main menu** by dragging the Info submenu icon from the Menus palette to OneButton's main menu and releasing it just below the title bar.

10. **Delete the Preferences and "Info..." menu items and the Edit submenu** by highlighting and then pressing the Delete key for each one.

11. **Enable the Info Panel menu item in OneButton's Info submenu** by highlighting it, turning off the disable switch in the **MenuCell Inspector** and clicking **OK**.

    Enable the Help menu item in a similar fashion.

12. **Add a customized Info panel** by performing steps 13-16 below.

13. **Open the Windows palette** by clicking on the second icon (from the left) at the top of the Palettes window.

14. **Add an Info panel** by dragging an **Info** icon from the Windows palette and releasing it in the Workspace just below the OneButton App window. Note the new **icon (object) called "Info" in the File Window**.

15. **Make a copy of the CHI_Icon appear in the Info panel** by dragging its icon from the Icons window on top of the generic icon (actually a disabled button) in the Info panel and releasing it.

16. **Customize the Info panel text** by editing it with single- and double-clicks and keyboard entry.

17. **Make a connection** between the **Info Panel menu item and the Info Panel** by control-dragging from the item to the panel's title bar (note the **black line**) and double-clicking the "**makeKeyAndOrderFront**:" **method** in the MenuCell Inspector (note the dimple).

    When the Info menu item is chosen an **action message** will be sent to the Info panel to become the **key window** and appear **in front of all other windows**.

18. **Add a customized Help panel** (in a similar fashion to the way the Info panel was added) by performing steps 19-31 below.

19. **Add a new panel** by dragging the **Panel** icon from the Windows palette into the WM. Note the new **icon called "Panel" in the File Window**.

20. **Change the title of this new panel to "Help"** by dragging to "Attributes" in the pop-up-list in the Window Inspector, and typing the new title.

21. **Force the Help panel to show up when OneButton is launched** by turning on the "**Visible at Launch Time**" switch in the Window Inspector and then clicking **OK**.

22. **Add the CHI_Icon to the Help panel** by rubber-banding (press and drag) around the CHI_Icon in the Info Panel, choosing **Copy** from IB's Edit menu, clicking in the Help panel, and then choosing **Paste** from IB's Edit menu. Drag the new CHI_Icon to the top left of the panel.

23. **Add customized text to the Help panel** by performing steps 24-26 below.

24. **Open the Basic Views palette** by clicking on the third icon (from the left) at the top of the Palettes window.

25. **Add text to the Help panel** by dragging the **Text** icon from the Basic Views palette and releasing it in the Help panel.

26. **Change this text to "Press the Button!"** by double-clicking on **"Text**" and typing the new text.

27. **Make the text larger** by repeatedly clicking **Larger** in IB's Format menu.

28. **Play with the attributes of the text in the TextField Inspector** (e.g. turn off the **editable switch**) and then click **OK.**

29. **Make the Help panel smaller** by performing steps 30-31 below.

30. **Make the Help panel's resize bar appear** (only within IB) by clicking on the small **resize button** at the top left corner of the panel.

31. **Make the panel smaller** by dragging on one of the 3 parts of the resize bar. Click on the resize button again to use another part of the resize bar.

32. **Make a connection** between the **Help menu item and Help panel** by control-dragging from the item to the panel's title bar and double-clicking on the "**makeKeyAndOrderFront**:" **method** in the MenuCell Inspector.

33. **Rearrange OneButton's window and panels** so that they don't overlap and the Help panel is in the middle. Do this by dragging on their title bars.

34. **Change the Application icon** (which will eventually appear in the Workspace) by performing step 35 below.

35. **Open the Project Inspector** by dragging to **Project** in the pop-up list at the top of the Inspector window. If **Files** is highlighted (in black) on the "**Attributes Files**" button just below the pop-up list, then click on this button so **Attributes** is highlighted.

    Make sure the **generic icon** is highlighted, click on the **Set** button (an Open panel opens) and double-click on the **CHI_Icon.tiff** file in the Open panel. The CHI_Icon appears in the Project Inspector.

    The screen should look much like the **"OneButton With Icons in IB" screen dump** on page 30.

36. Save your work by choosing **Save** from File submenu (5 files are updated).

37. **Create a new executable file OneButton.debug** by choosing **Make** from the File submenu (or use the key-equivalent Command-k).

38. **Quit Interface Builder** (so the screen is simpler).

39. **Highlight the "OneButton.debug"** file name in the File Viewer's Shelf. The **CHI_Icon appears**.

40. **Launch the OneButton.debug application** (by clicking on its icon) and note that the **Help panel appears but the Info Panel does not** (see Step 21 above). Test the button and menu items and then quit.

    See the **"OneButton with Application Icon in WM" screen dump** on page 31.

Go to page 32 for a wrap-up of Project 1B.

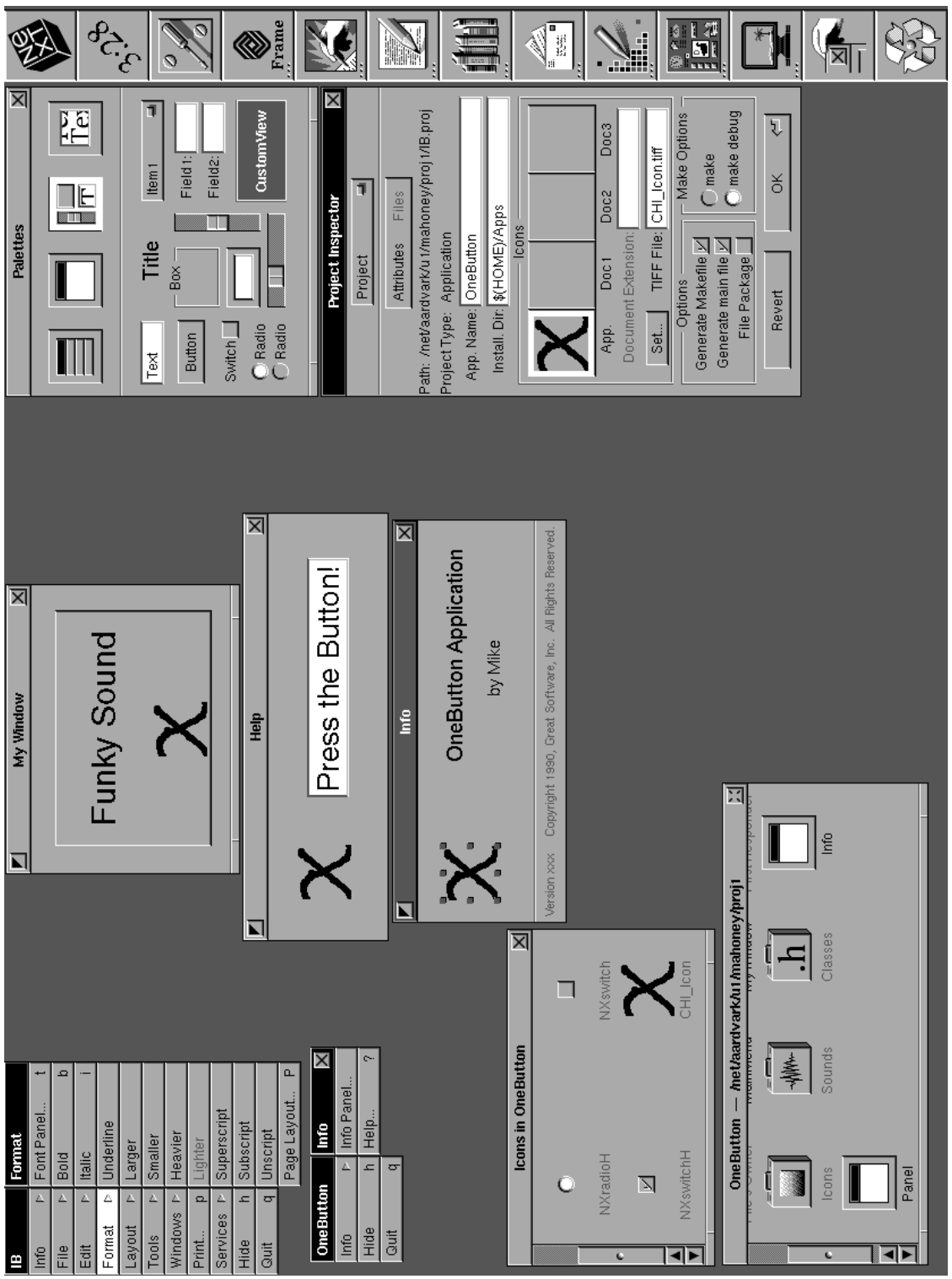
Performance Note: NeXT recommends that Info (and possibly) Help panels be placed in interface (.nib) files other than the main one in order to improve the performance of an application. Since an Info panel is rarely opened, there is little reason to "unarchive" it before the user requests to see it.

A separate interface file was not used here in the interest of simplicity.

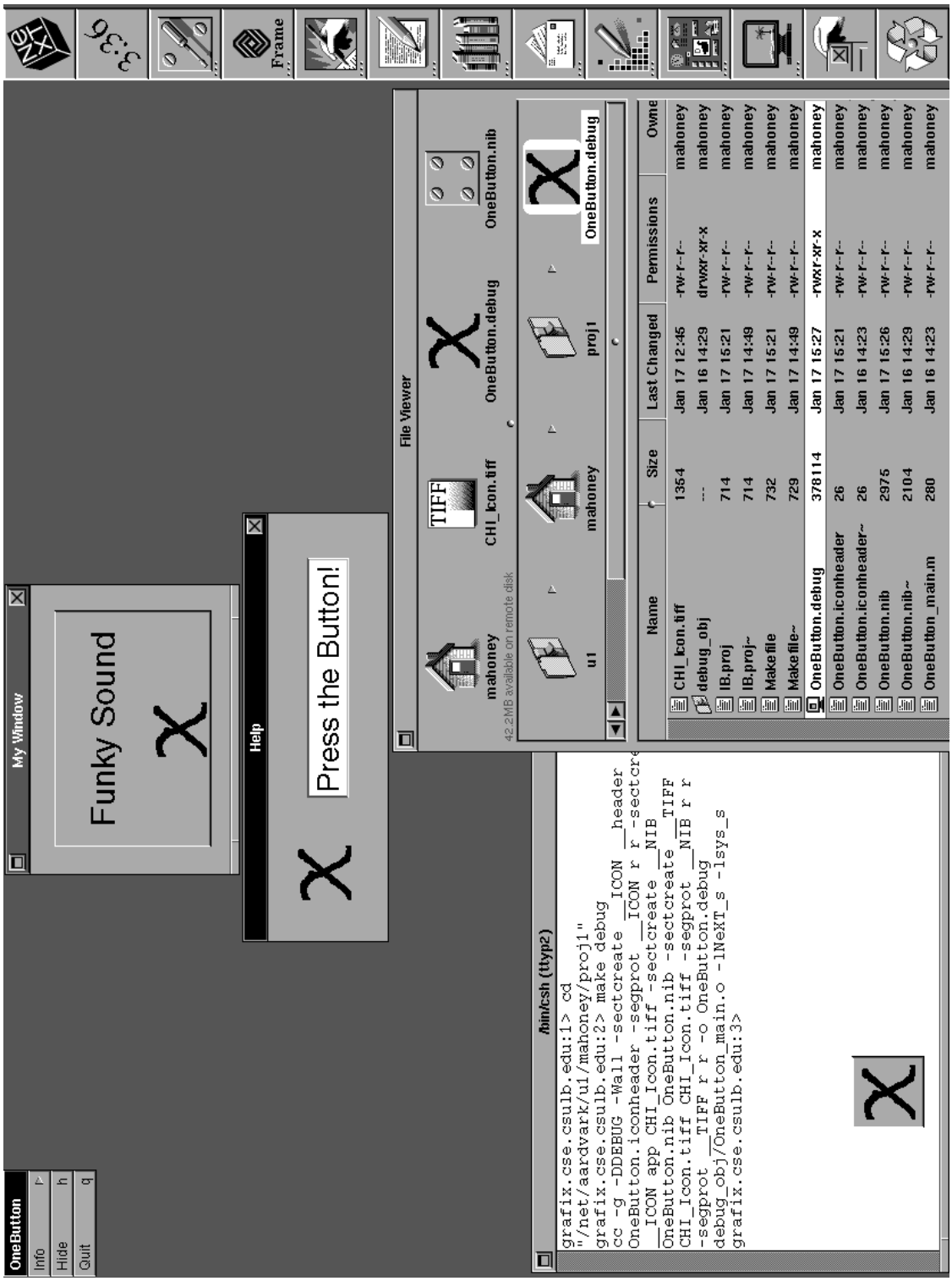
ScrollView Note: To get experience with a ScrollView object you can replace steps 25-28 with:

Click the right button at the top of the Palettes window to open the **Scrolling Views palette** and drag a **ScrollView object** from it into the Help panel. Enter more than a page of text and customize it using the ScrollView Inspector.

OneButton With Icons in IB



OneButton with Application Icon in WM



Project 1B Wrap-up

We

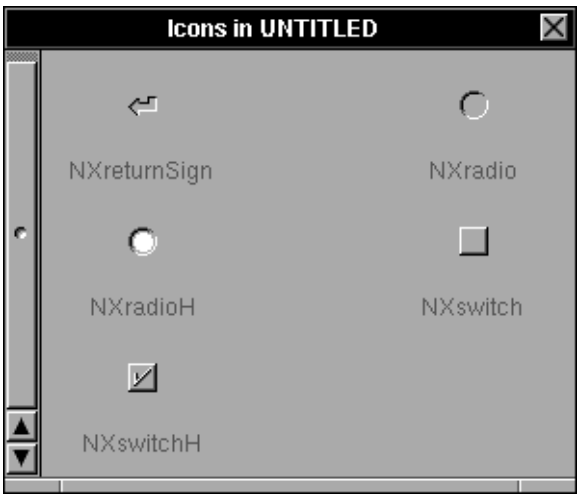
- added a previously created icon to the project
- added and customized two panels with the icon and text
- added two menu items
- connected the menu items to the panels so that the panels would appear when the menu items are chosen
- added an application icon which shows up in the Workspace and identifies the application
- learned how to add and customize a new text object to a panel.

We have not yet created an application with a computational engine.

We'll do this in the Project 2 after discussing the Palettes, Icons, and Inspector windows.

The Icons Window

- Double-clicking on the suitcase icon labeled “Icons” in the File window opens the Icons Window below



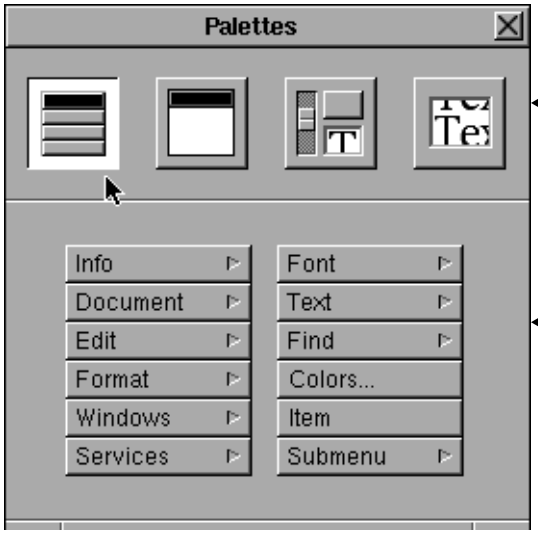
Icons readily available for the application’s interface.

- Other icons may be dragged into this window and thus be made available for use in the interface being built in IB.
- Custom icons can be created using the **Icon application** in the /NextDeveloper/Demos folder.
- To inspect one of these icons, double-click on it and use the **Icon Inspector**.
- To make an icon appear on top of a button in an interface, drag it from the Icons window and release it on top of the button.
- **Application icons** are set using the Project Inspector. They must be 48 x 48 pixels in size and have alpha channels (see the Icon application).

The Palettes Window

- contains 4 palettes which give you access to the Application Kit's primary user interface objects.
- The palette that is visible is determined by clicking one of the four buttons at the top of the window.

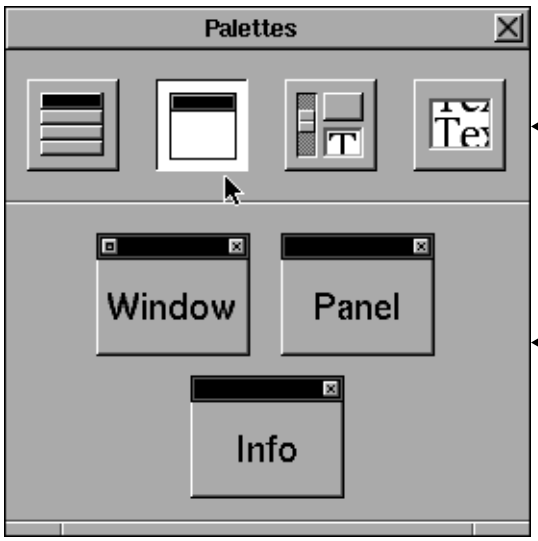
Menus palette



Left button highlighted

These objects may be dragged anywhere within a menu or submenu in the application's interface.

Windows palette

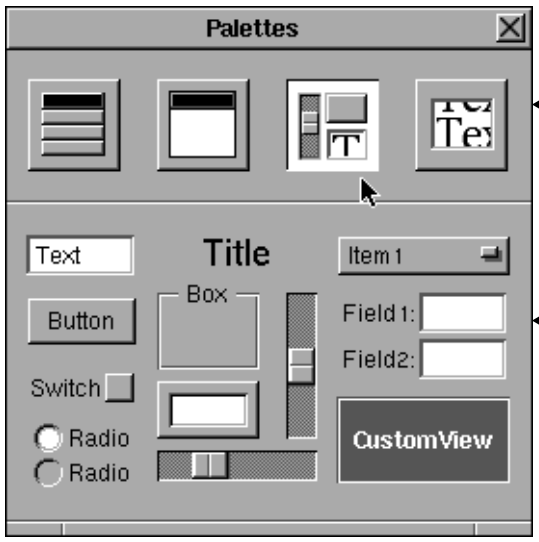


Second button highlighted

These objects may be dragged anywhere within the Workspace and will expand to a window or panel.

The Palettes Window (cont.)

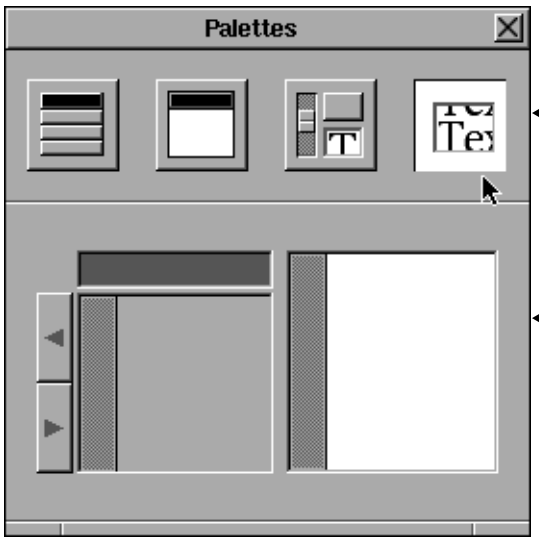
Basic Views palette



Third button highlighted

These objects may be dragged anywhere within a window or panel in the application's interface.

Scrolling Views palette



Right button highlighted

These two objects (NXBrowser and ScrollView) may be dragged anywhere within a window or panel in the application's interface.

Additional palettes may be added with custom subclasses of View. See Project 6 (**SketchPalette**).

The Inspector Window(s)

The Inspector window is the location where nongraphic features of the objects in the application are edited.

The display inside the Inspector window changes often and usually depends on which object in the interface is selected. It can also be changed explicitly by choosing one of the 6 items in its pop-up list.

The display falls into one of the following 6 groups.

- **Attributes -** changes depending on the object selected; mainly for editing nongraphic attributes
- **Connections -** for viewing and setting up connections for outlets and action messages
- **Autosizing -** where you can specify how an object will respond when its window or superview is resized
- **Miscellaneous -** place where you set the location and dimensions of the selected View or Window
- **Class -** for viewing and editing outlets and actions
- **Project -** see the next page

See the NeXTstep Concepts manual for details.

Several different types of Inspector window displays can be seen in the screen dumps in this tutorial.

The Project Inspector

opens when **Project** is chosen from IB's File submenu or from the Inspector window's pop-up list.

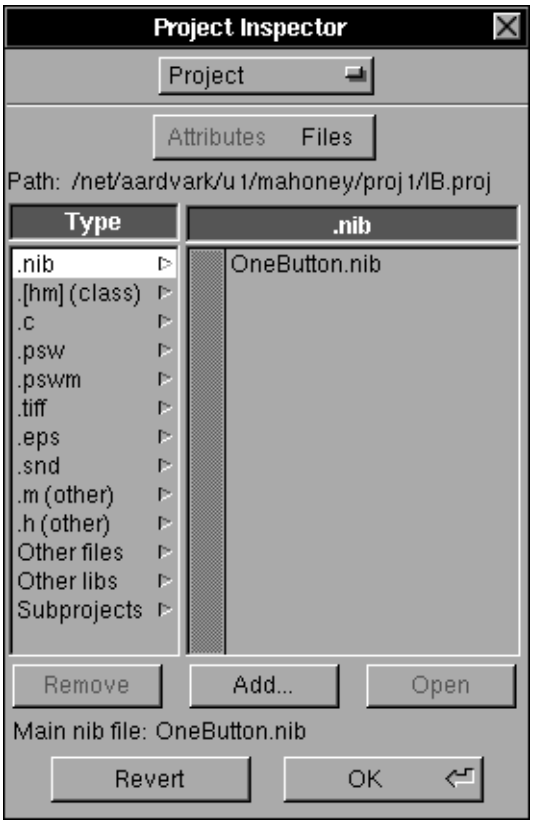
It contains two windows, one for the **Attributes** of the project and one for the **Files** of the project.

Application icons and **document icons** (e.g. icons for ".wn" documents for WriteNow) can be set up in the Attributes Project Inspector.

Below we see the two Project Inspector windows for **OneButton**.

Attributes

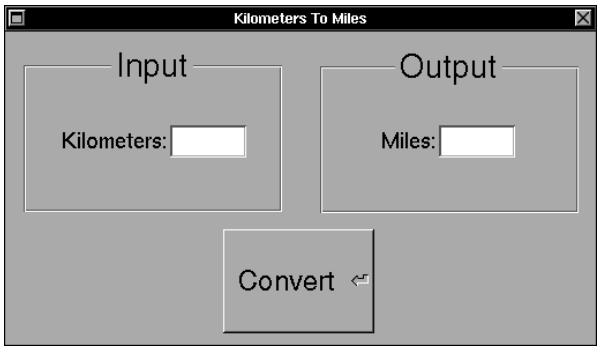
Files



Project 2 - KmToMi

(Convert Kilometers to Miles)

KmToMi is a simple application whose interface consists of a menu and a window which contains an **input area**, an **output area** and a **button**. It will look like:

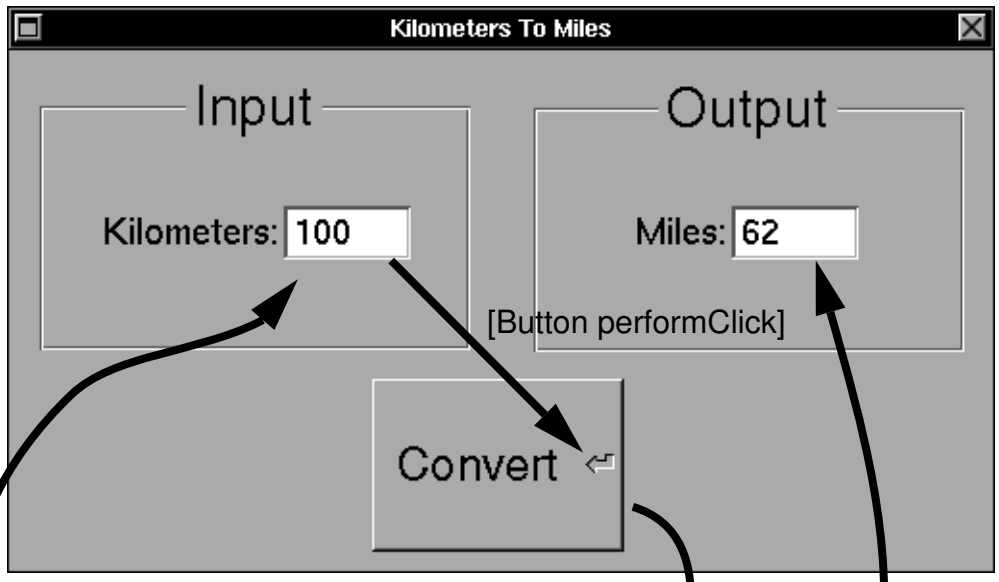


When the button is pressed it will send an **action message** to a **custom object** (the **target**) to invoke a **method** (procedure). This method will then read a kilometer value from the input text area (called a Form object), convert it to miles, and send (write) the result to the output text area (another Form). See the next page.

The **main objectives** of Project 2 are to

1. create an application with a computational engine
2. learn how to use Interface Builder to create a **subclass** and **instantiate** it, make **connections** between objects, and use and customize more user interface objects
3. understand the process of editing an Objective-C Language file containing a class implementation.

The 4 Messages in KmToMi




```
// ConvertObject.m
// Generated by IB (except for 4 lines)
#import “ConvertObject.h”
#import <appkit/Form.h>    // inserted

@implementation ConvertObject
- convertMethod:sender
{
 float miles;     // inserted
                          // read input and convert
 miles = ( 0.62 * [kmOutlet floatValueAt:0] );
 [miOutlet setFloatValue:miles at:0];
 return self;       // send output and return
}
@end       // see the ConvertObject.h file on page 43
```

The general format of a **message** is

[receivingObject methodName:arguments]

Project 2 Step by Step

1. **Launch Interface Builder** (IB) and choose **New Application** from its File submenu.

2. Choose **Save As** from the File submenu and save as **"proj2/KmToMi"** ("Kilometers to Miles" in the newly created proj2 folder). Choose **Project** from the File submenu and create a project as before.

3. **Add an input area interface** to the application's main window by performing steps 4-8 below.

4. Drag a **"Box" icon** from the Basic Views palette into the top left corner of "My Window" and then drag on its handles to make it larger. (The "Box" object is purely cosmetic.)

5. **Rename the box to "Input"** by double-clicking on "Box" and typing the new name. Then make "Input" larger by repeatedly choosing **Larger** from IB's **Format** submenu.

6. Drag a **Form icon** ("Field 1:", "Field 2:" etc.) from the Basic Views palette into the "Input" box. This icon actually contains two Form objects, each consisting of a title and a (white) text area.

7. We will only need one of the Forms so **Alt-drag** upward on the icon's bottom middle handle to **get rid of the "Field 2:" Form**. (Dragging downward will create more Forms, if needed.)

8. **Rename the Form field to "Kilometers"** by double-clicking on "Field 1:" twice and entering the new name. Then **make the Form text larger** by repeatedly clicking on **Larger** in IB's Format submenu.

9. **Add an output area interface** to the application's main window by performing steps 10-11 below.

10. **Make a copy of the Input area interface** (Box and Form) by dragging (**rubber-banding**) a rectangle around it and choosing **Copy** and then **Paste** from IB's Edit submenu (or use the **key-equivalents Command-c** and **Command-v**).

11. Drag the copy to the right and rename the copied Box to **"Output"** and the copied Form to **"Miles"** (use double-clicks).

12. **Add a button to** "My Window" as in Project 1A. **Enlarge** the button and its text and rename it **"Convert"**.

13. **Customize the application's main window** ("My Window") by performing steps 14-15 below.

14. **Make "My Window" smaller** using its resize bar.

15. Drag to **Attributes** in the Inspector's pop-up list, rename "My Window" to "Kilometers to Miles", turn off the **"Resize Bar" switch** and then click **OK**.

16. **Create a subclass "ConvertObject" of the class "Object"** by performing steps 17-20 below.

17. **Open the Classes window** by double-clicking on the **Classes suitcase icon** in the File window.

18. **Select "Object**" in the Classes window browser by scrolling to the far left and clicking on "Object".

19. Drag to **Subclass** in the **"Operations" pull-down list**. Note that a new class **"MyObject"** appears.

20. **Rename this new subclass "ConvertObject"** by double-clicking on "MyObject" under the icon well in the Classes window and entering the new name.

21. **Create an instance object of the ConvertObject class** by dragging to **Instantiate** in the "Operations" pull-down list.

    Note the **new icon labeled "ConvertObjectInstance"** in the File window. We can now make connections with this icon (object).

22. **Add outlets called "kmOutlet" and "miOutlet" to the ConvertObject class** by performing steps 23-24 below.

    These **outlets** will be made to point to the Form objects so that "ConvertObject" can send them messages.

23. **Open the Class Inspector** by dragging to **Class** in the Inspector's pop-up list.

24. Highlight (in black) **Outlet** on the **"Outlet Action" button**, click in the white text area, and enter "**kmOutlet**" and then "**miOutlet**". (Outlet names should begin with a lower case letter.)

25. **Make a connection** between the **"ConvertObjectInstance" icon (object) and the "Kilometers" Form object** by control-dragging from the icon to the Form (note the direction) and then double-clicking on the "**kmOutlet**" outlet in the CustomObject Inspector.

Make sure that the **gray** (not black) rectangle surrounds the Form before releasing the mouse button. Read the **help messages** near the bottom of the CustomObject Inspector. See the **"Outlet Connection" screen dump** on page 44.

26. **Make a connection** between the **"ConvertObjectInstance" icon (object) and the "Miles" Form object** in a similar fashion. Double-click on "**miOutlet**" in the CustomObject Inspector.

27. **Add a method called "convertMethod:" to the (target) ConvertObject class** by performing steps 28-29 below.

    This method will be set up to perform an action in response to an **action message** received from the "Convert" button following a click.

28. **Open the Class Inspector** by dragging to the **Class** item in the Inspector's pop-up list.

29. Highlight (in black) **Action** on the **"Outlet Action" button** and enter **"convertMethod:"**. (Method names should begin with a lower case letter.)

30. **Make a connection** between the **"Convert" button and the "ConvertObjectInstance" icon** (object) by control-dragging from the button to the icon (note the direction) and double-clicking on the **"convertMethod:"** method in the **Button Inspector** (note the dimple).

31. **Make a connection** between the **"Kilometers" Form and the "Convert" button** by control-dragging from the Form to the button and double-clicking on the **"performClick:" method** in the **FormCell Inspector**.

    When the Return key is pressed after typing in a number in "Kilometers" Form in the active application, an **action message** will be sent to the "Convert" button to perform as if it had been clicked.

32. **Add an icon to indicate the Return key functionality** by opening the Icons window (double-click the icons suitcase), dragging a **NXreturnSign icon** and releasing it on top of the "Convert" button.

33. **Create class interface** (ConvertObject.h) **and implementation** (ConvertObject.m) **Objective-C Language files** by dragging to **Unparse** in the "Operations" pull-down list in the Classes window.

    Make sure the "ConvertObject" class in the Classes window is highlighted first. Add these files to the project when prompted.

    See the **"Unparse and performClick" screen dump** on page 45.

34. Bring these two files into the **Edit application** by highlighting **Files** on the **"Attributes Files" button** in the **Project Inspector** and double-clicking on **"ConvertObject.[hm]"**.

35. Insert the Objective-C line `#import <appkit/Form.h>` after the line " #import "ConvertObject.h" " in the **ConvertObject.m** file.

36. **Insert the 3 Objective-C lines**

```
float  miles;
miles = ( 0.62 * [kmOutlet floatValueAt:0] );
[miOutlet  setFloatValue:miles  at:0];
```

before "return self;" in the "**convertMethod**:" method in the **ConvertObject.m** file.

See the **contents of the ConvertObject.m** on page 39.
See the **contents of the ConvertObject.h** at the bottom of this page.

37. Choose **Save** from Edit's File submenu and **Hide** from Edit's main menu.

38. Choose **Make** from IB's File submenu and save the interface file when prompted. **Quit IB.**

39. **Launch KmToMi** by clicking in the Shell window (to activate it) and then entering **KmToMi.debug**.

40. **Test KmToMi** by clicking in the "Kilometers" white text area, typing a real number and pressing **Return or** clicking on the **"Convert" button**. Try it a few more times and then **Quit** KmToMi.

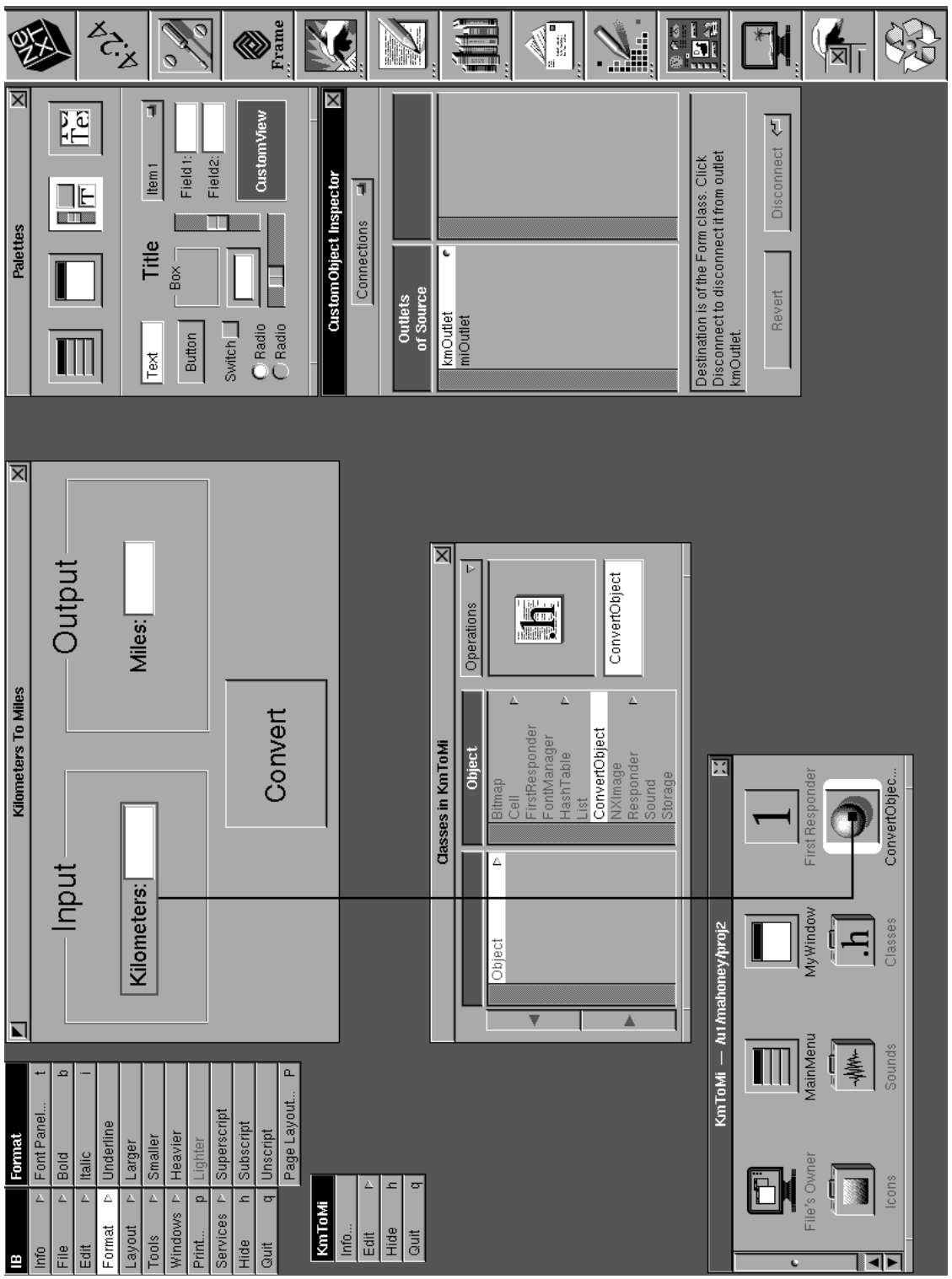
See the "**KmToMi in Workspace" screen dump** on page 46.

Go to page 47 for a wrap-up of Project 2.

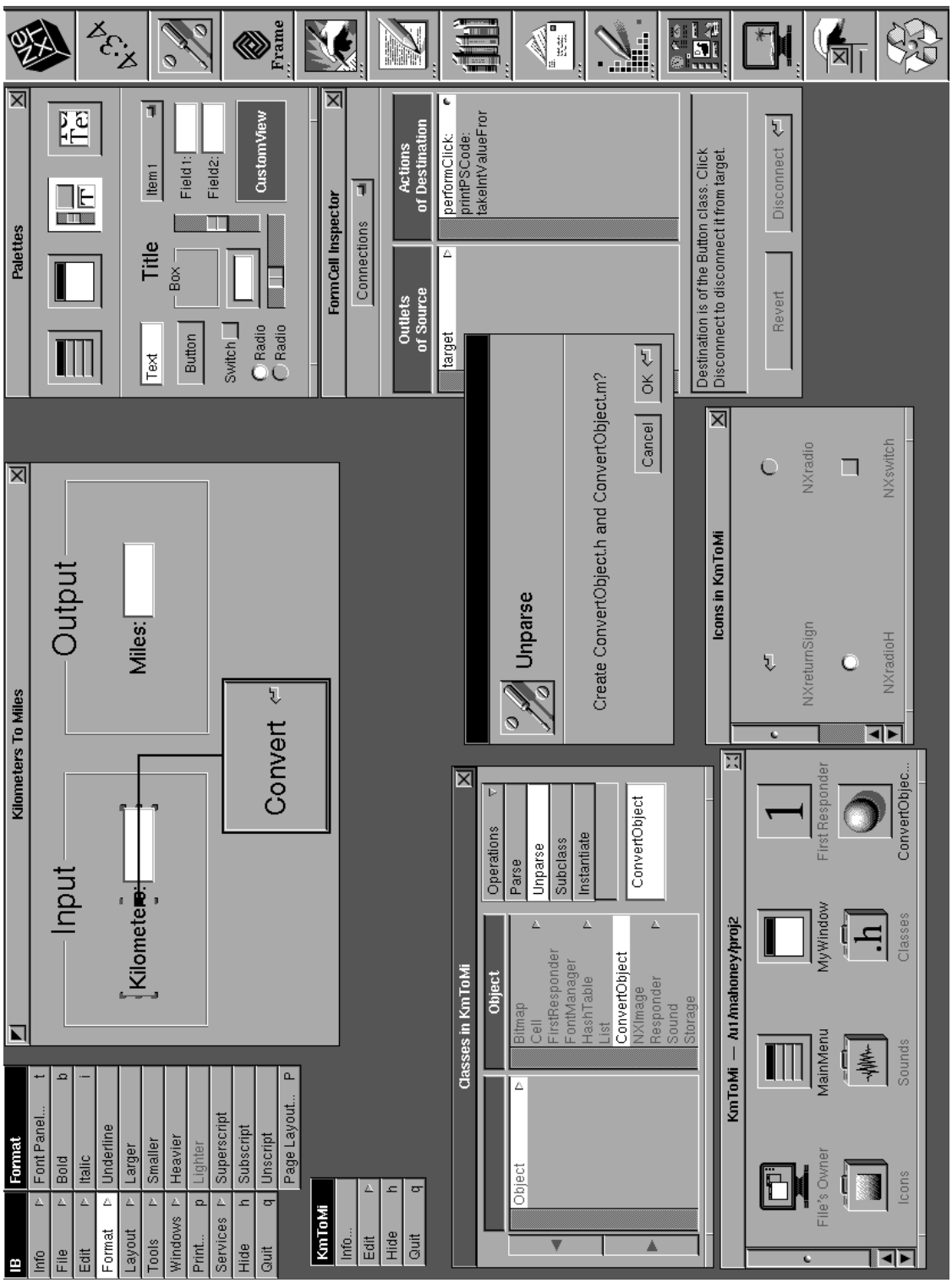
ConvertObject.h file

```
/* Generated by Interface Builder */
#import <objc/Object.h>
@interface ConvertObject:Object
{
 id   miOutlet;
 id   kmOutlet;
}
- convertMethod:sender;
@end
```

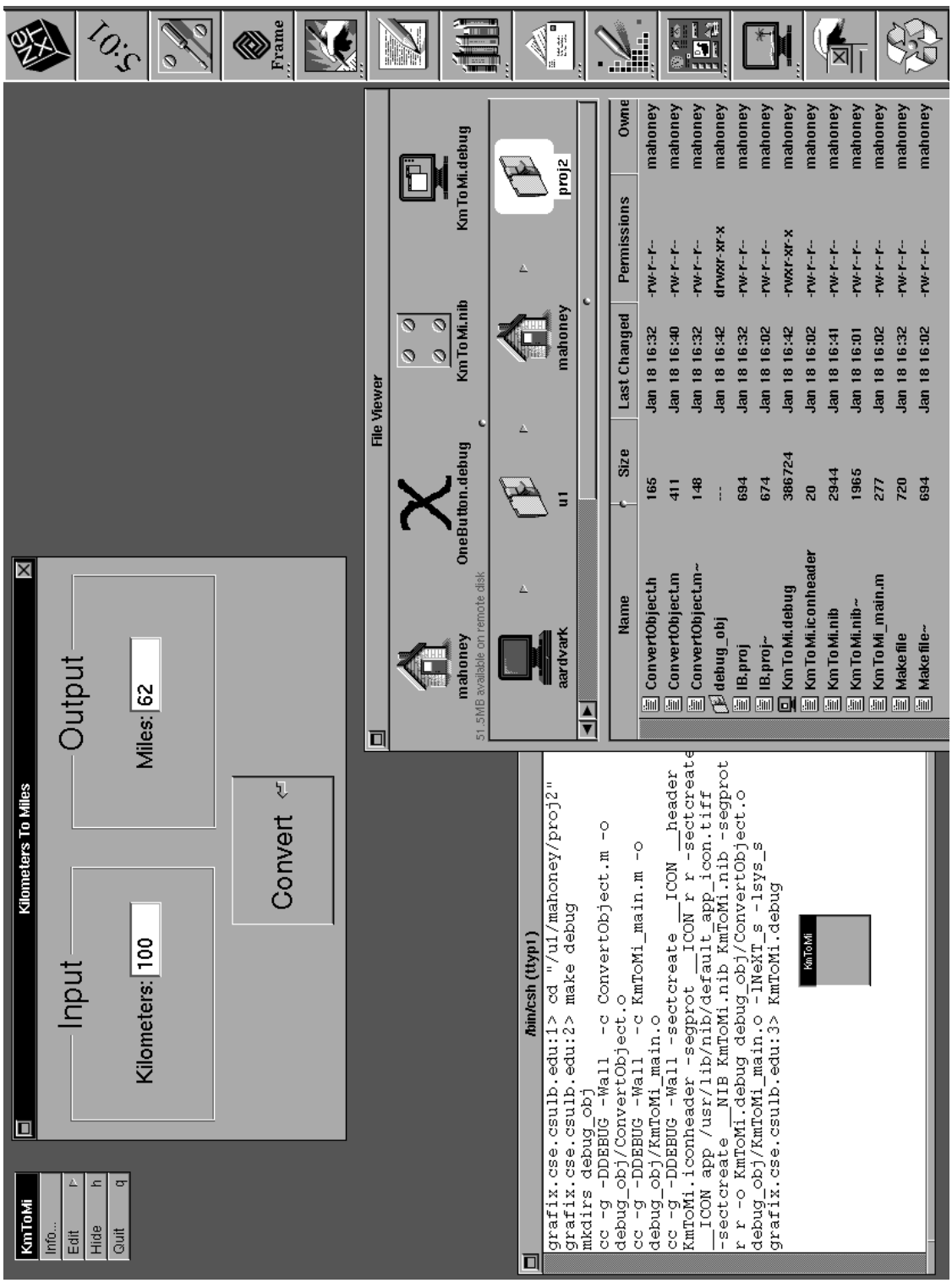
Outlet Connection (KmToMi)



Unparse and performClick (KmToMi)



KmToMi in Workspace



Project 2 Wrap-up

We

- created a simple interface with two forms, two boxes and a button
- created our own subclass of “Object” and instantiated it
- made “outlet” connections between objects (which set up outlet pointers from one object to another)
- made “target/action” connections between objects
- “Unparsed” the interface to create Objective-C Language class interface and implementation files
- “Edit-ed” the class implementation file by adding a computational engine, and
- compiled and tested the application.

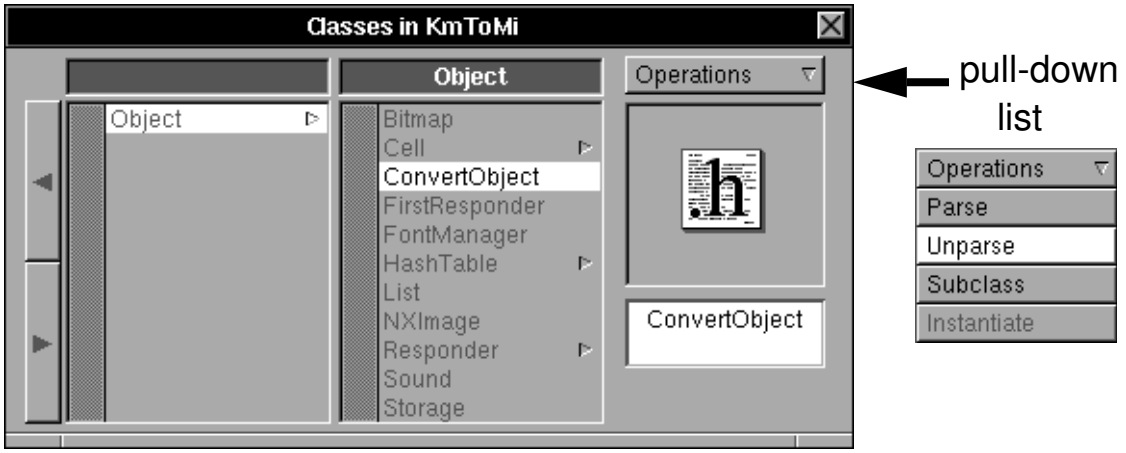
Warning: if you Unparse a second time, then the class files will be overwritten and your code may be lost.

In Project 3 (on page 51) we’ll see how to do special drawing in a window.

First we discuss the Classes window, Operations from its pull-down list, and the application development process.

The Classes Window

- Double-clicking on the suitcase icon labeled “Classes” in the File window opens the Classes Window below



- By clicking on the scroller arrows you can see the **inheritance hierarchy** of the **Application Kit objects** available to an interface in IB.
- To inspect one of these classes, double-click its name and look in the **Class Inspector.**
- The pull-down list titled **“Operations”** contains four significant items which are discussed on the next page.

Operations on Classes

(accessible from the pull-down list labeled "Operations" in the Classes window)

1. **Subclass -** defines a subclass of the selected class

   Usually only the **"Object"**, **"View"** or **"Application"** classes will need to be "subclassed".

   A **subclass of "Object"** is generally used to contain the code which controls the application and/or the code for the computational engine.

   A **subclass of "View"** is generally used to contain code for special drawing (graphics) in a window.

   For **subclasses of "Application"** see **Yap** or **Draw**.

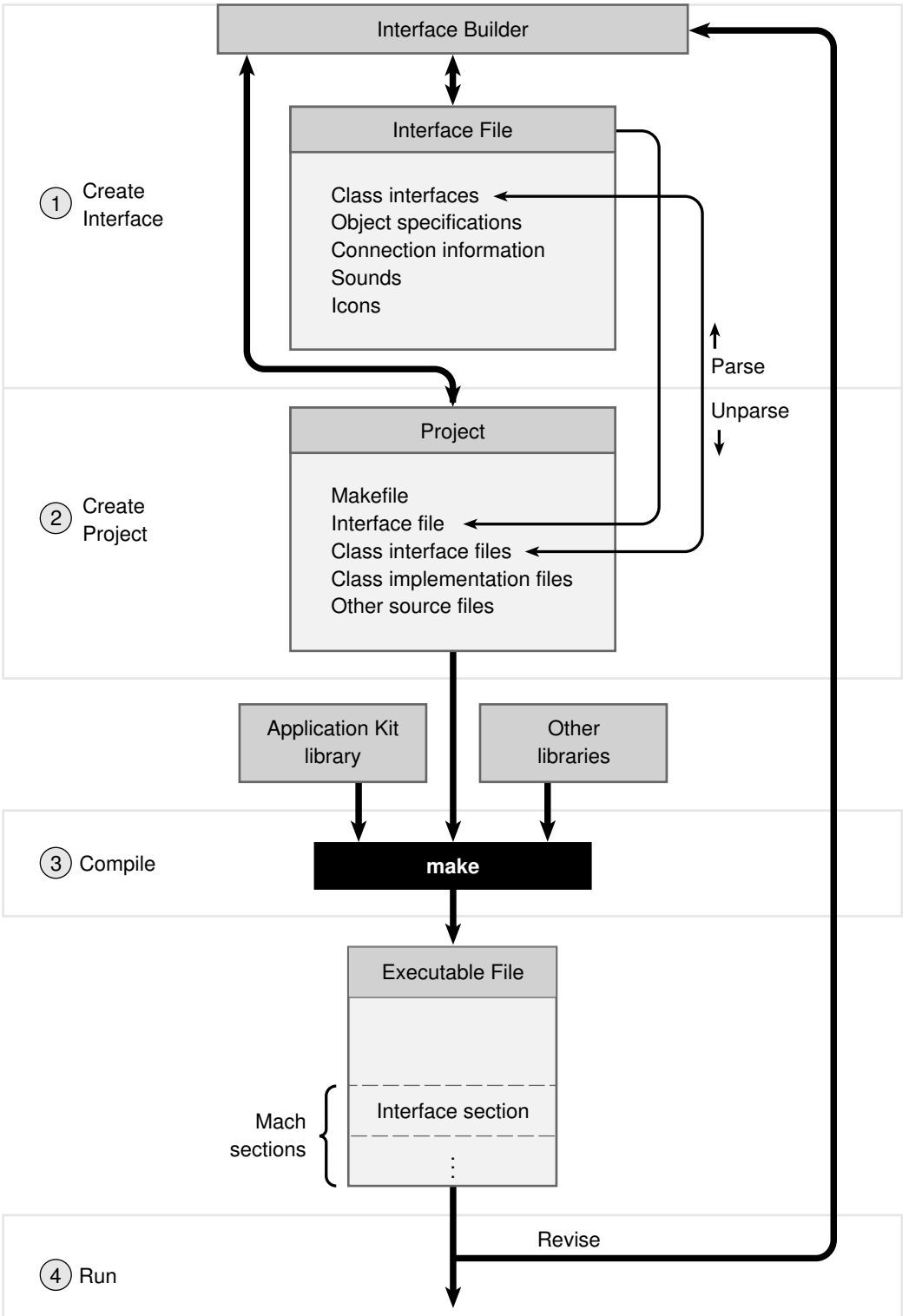
2. **Instantiate -** creates an object of the selected class and places its icon in the File window.

   This allows you to make connections with this object. This operation is disabled for "View" subclasses. Instead, drag a **"CustomView" icon** from Basic Views palette and use the Inspector window.

3. **Unparse -** generates **class interface (.h)** and **skeletal class implementation (.m) files** based on the class highlighted in the Classes window

4. **Parse -** for reading the class definition from an interface (.h) file. Do Project 5 to see how it works.

The Application Development Process

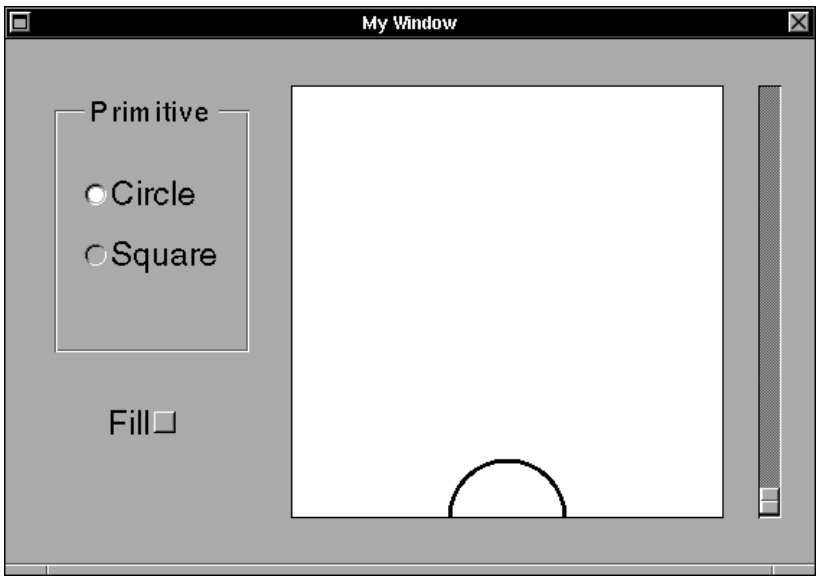


Courtesy of NeXT Computer, Inc.

Project 3 - SimpleGraphics

(display a circle or square which moves with a slider in a custom View object)

When launched, SimpleGraphics main window will look like



The **objectives** of are to learn how to use IB to

- add a custom “View” object (for special drawing)
- add radio buttons , a switch button and a slider
- make connections between the interface objects and the custom View object.

We’ll also learn more about Objective-C Language and Display PostScript® graphics functions.

Project 3 Step by Step

1. **Launch IB** and choose **New Application**, **Save As** and then **Project** from its File submenu. Save as "**proj3/SimpleGraphics**".

2. **Create a subclass of the View class** called "**SimpleView**" by opening the Classes window, highlighting **"View**" in its browser, dragging to **Subclass** in the "Operations" pull-down list and renaming "MyView".

3. **Add a custom "View" object of the "SimpleView" class** by performing steps 4-5 below.

4. Drag a **"CustomView" icon** from the Basic Views palette into "My Window".

5. **Change the class of the "CustomView" icon (object) to "SimpleView"** by dragging to **Attributes** in the Inspector's pop-up list and then double-clicking on **"SimpleView"**.

   Note that the name of the "CustomView" icon changes to "SimpleView". **We have actually instantiated an object of the "SimpleView" class. "View" objects cannot be "Instantiate(d)" via the "Operations" pull-down list**.

6. **Make "SimpleView" 300 x 300 pixels square** by dragging to **Miscellaneous** in the **CustomView Inspector**, changing the width and height to 300 and pressing Return.

7. Drag "SimpleView" so that there's some room on the left side of "My Window" for some radio buttons and on the right side for a vertical slider. Resize "My Window" appropriately.

8. **Add a "boxed" pair of customized radio buttons** to the interface by performing steps 9-12 below.

9. Drag a **"Box" icon** from the Basic Views palette to the left of "SimpleView". Make it and its text larger and rename it "**Primitive**".

10. Drag a **"Radio Radio" icon** from the Basic Views palette into the "Primitive" Box.

11. **Spread the 2 buttons out** by dragging (without the Alternate key) on the handles of the "Radio Radio" icon and make the text larger.

    (Using the Alternate key while dragging on these buttons will create more of them.)

12. **Rename the buttons "Circle"** and **"Square"** by double-clicking on the top "Radio", typing "Circle", pressing the **Tab** key and typing "Square".

13. **Add a "fill toggle" switch** to the interface by dragging a **"Switch" icon** from the Basic Views palette below the "Primitive" Box. Make its text larger and rename it **"Fill"**.

14. **Add a "Y-location" Slider** to the interface by performing steps 15-17 below.

15. Drag a **vertical Slider object** from the Basic Views palette into "My Window" to the right of "SimpleView".

16. **Make the slider the same height as "SimpleView"** by dragging to **Miscellaneous** in the Slider Inspector, changing its height to 300 and pressing Return.

17. **Make the slider's range be 0-300** and its **current (initial) value be 0** by dragging to **Attributes** in the Slider Inspector and entering the values.

18. **Add the 4 action methods "displayCircle:", "displaySquare:", "fillToggle:" and "sliderMoved:"** to the **SimpleView class** by performing steps 19-20 below.

19. Click "SimpleView" in "My Window" and then drag to **Class** in the **CustomView Inspector**'s pop-up list.

20. Highlight **"Action"** on the "Outlet Action" button and enter the 4 method names in step 18 above.

21. **Make a connection** between the **"Circle" radio button and "SimpleView**" by performing steps 22-23 below.

22. Double-click on the "Circle" radio button so that it's highlighted (in dark gray).

23. Control-drag from the "Circle" radio button to "SimpleView" and then double-click on the **"displayCircle:"** method in the **ButtonCell Inspector**.

    If a SimpleGraphics user clicks the "Circle" radio button, then the button will send an action message to SimpleView which invokes "displayCircle:".

    See the **"SimpleGraphics in IB" screen dump** on page 55.

24. **Make a similar connection** between the **"Square" radio button and "SimpleView**". Double-click on the **"displaySquare:"** method.

25. **Make a connection** between the **"Fill" switch and "SimpleView**" by clicking on the switch, control-dragging from the switch to "SimpleView" and then double-clicking on the "**fillToggle**:" method in the Inspector.

26. **Make a connection** between the **slider and "SimpleView**" by clicking on the slider, control-dragging from the slider to "SimpleView" and then double-clicking on the "**sliderMoved**:" method in the Inspector.

27. **Create class interface** ("SimpleView.h") and **class implementation** ("SimpleView.m") **files** by dragging to **Unparse** in the "Operations" pull-down list in the Classes window. Add these two files to the project.

28. **Bring these two files into the Edit application** by double-clicking on "**SimpleView.[hm]**" in the "Files" window of the Project Inspector.

29. **Insert the 3 instance variable and "initFrame" and "drawSelf" method declarations in the** "**SimpleView.h**" **file.** See page 57 for details.

30. **Insert the code specified** on pages 58-60 **in the "SimpleView.m" file.** In particular, insert the complete "initFrame" and "drawSelf" methods and add the "guts" of the "displayCircle:", "displaySquare:", "fillToggle:" and "sliderMoved:" methods (or get the files via e-mail from Mahoney).

31. **Save the two files** and **quit Edit**.

32. Activate IB, choose **Save** and then **Quit IB**.

33. Open a shell window, enter "cd ~/proj3", "make" and then "SimpleGraphics".

34. Click near the middle of the slider, click on the "Square" radio button and then click on the "Fill" switch.

    See the **"SimpleGraphics in Workspace" screen dump** on page 56. **Note the output from the "printf" statements in the shell.**

35. Quit SimpleGraphics.

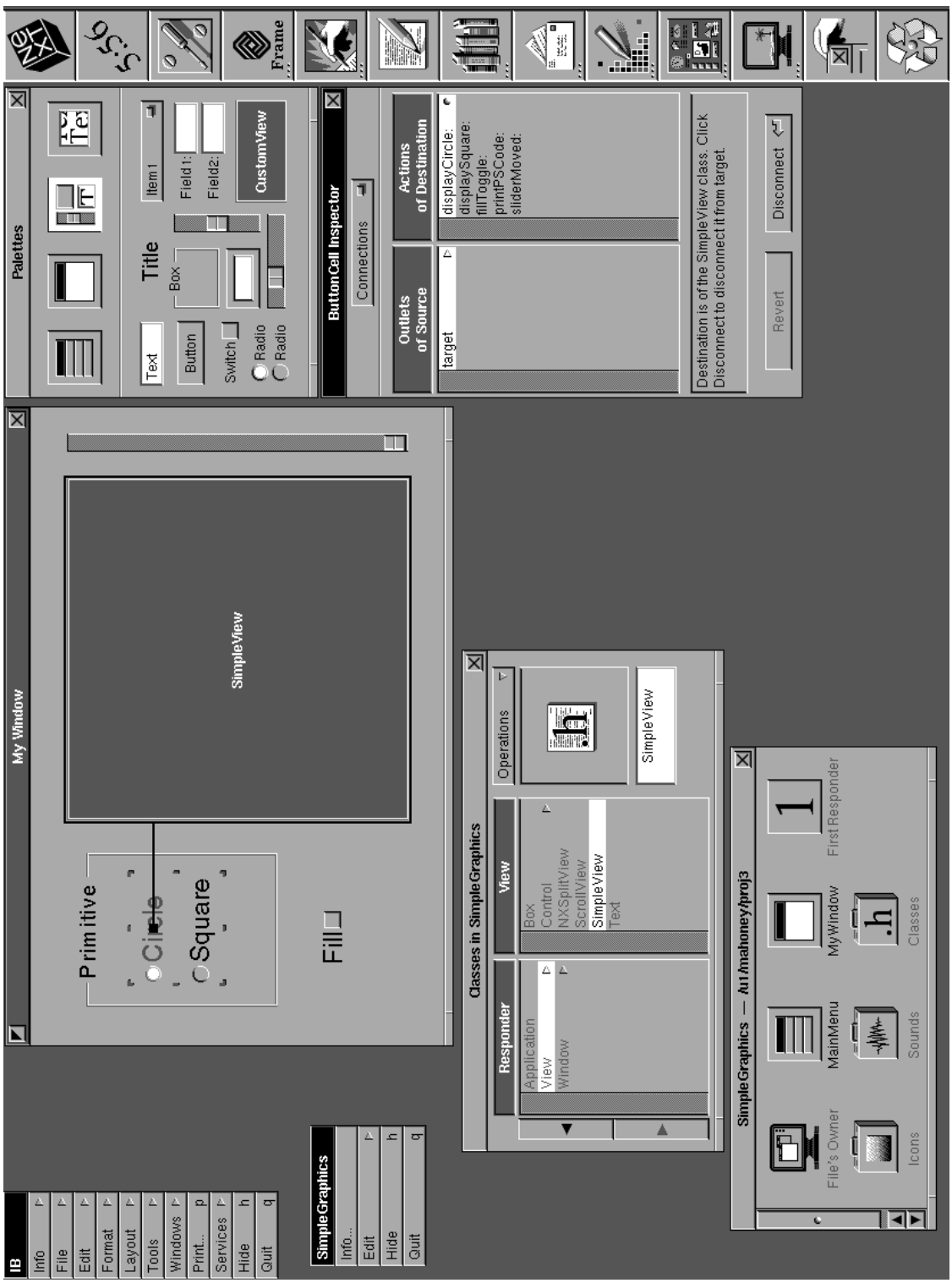
36. **Add the capability to print SimpleView's contents** by performing steps 37-38 below.

37. **Open the Menus Palette**, **drag the menu item labeled "Item"** into SimpleGraphics main menu, and rename it "Print...".

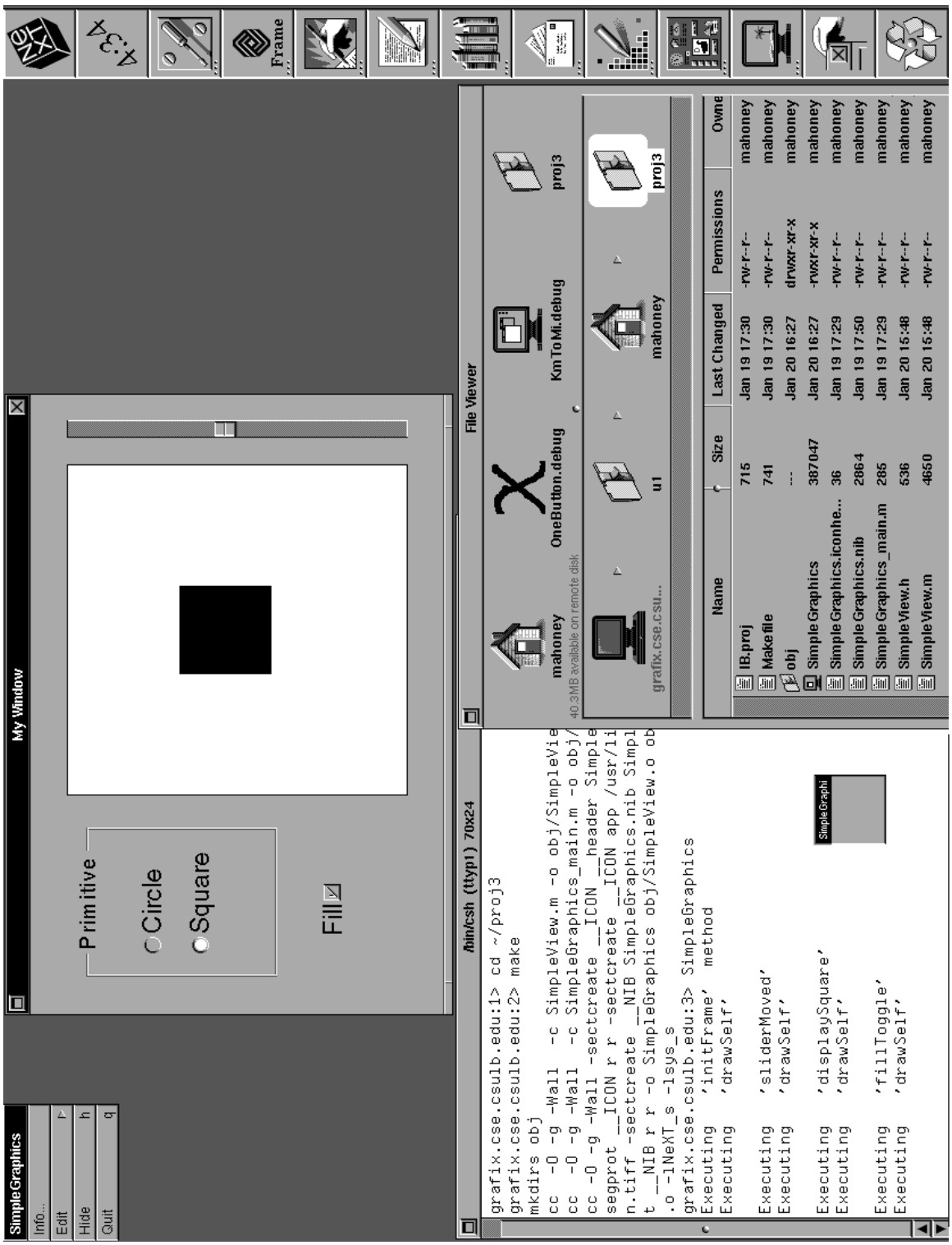
38. **Connect the new menu item to SimpleView** by control-dragging from it to SimpleView and double-clicking "printPSCode:" in the Inspector window.

39. **Save, make, execute and test SimpleGraphics** (in particular **"Print..."**).

SimpleGraphics in IB



SimpleGraphics in Workspace



SimpleGraphics Objective-C Files

SimpleGraphics_main.m

```
// Generated by NeXT's Interface Builder (except for the comments) when "Project" chosen.
                                        // the code in this file should not be edited
#import <stdlib.h>                      // required for the UNIX exit() function
#import <appkit/Application.h>          // required for the "Application" object (which
                                        //         will be the interface file's owner)
void main(int argc, char *argv[])       // every (Objective-) C program starts with main()
{
 NXApp = [Application new];             // create a new Application object and give it a name
 [NXApp loadNibSection:"SimpleGraphics.nib" owner:NXApp];       // load interface section
 [NXApp run];           // start event loop                                // from the executable
 [NXApp free];          // free the objects following a user's choice to Quit the app
 exit(0);
}
```

SimpleView.h

```
// Generated by IB during "Unparse" (except for the instance vars and the last two methods).
#import <appkit/View.h>                 // required because we're subclassing the View object

@interface SimpleView:View              // "SimpleView" is a subclass of "View"
{       int  primitiveToDisplay;        //  determines which graphics primitive is displayed
        int  fillFlag;                  //  determines whether primitive is filled or not
        float  sliderYvalue;            //  determines the Y position of the primitive
}
//  TARGET-ACTION METHODS               //   these were generated by IB
- displayCircle:sender;;                //   invoked when "Circle" radio button clicked
- displaySquare:sender;                 //   invoked when "Square" radio button clicked
- fillToggle:sender;                    //   invoked when "Fill" switch clicked
- sliderMoved:sender;                   //   invoked when Slider moved

//  OVERRIDDEN VIEW METHODS             //   these were inserted
- initFrame:(const NXRect *) frameRect; //   initializes SimpleView
- drawSelf:(NXRect *)rects :(int) rectCount;    //   contains PostScript drawing functions
@end
```

SimpleView.m

```
// 2/1/91
// The skeleton was generated by Interface Builder when "Unparse" was chosen.
// The rest was inserted by Mike Mahoney.
// The lines that were inserted can be discovered by reading the comments.

// The "printf" lines are for learning purposes only. They show that the "initFrame" method
//          is automatically invoked at launch time and that the "drawSelf" method is invoked as
//          a result of the "display" method being invoked. Do not invoke "drawSelf" directly.
// Launch "SimpleGraphics" from a shell window to see the output from "printf" statements.
                                        //          (e.g. see the shell output on page 56)

#import "SimpleView.h"                  //  generated by IB

#import <appkit/Slider.h>               //  inserted because message sent to slider
#import <dpsclient/wraps.h>             //  inserted for PS and NX graphics functions
                //  #import <appkit/appkit.h>  could be used here to include all App kit
                //          interface (.h) files but would slow compilation considerably

@implementation SimpleView

#define   CIRCLE  1                     //   #define(s)  and  declarations  inserted
#define   SQUARE  2

                //  initFrame is automatically invoked when SimpleGraphics is launched.
-initFrame:(const NXRect *) frameRect           //  this entire method was inserted
{
        printf ("Executing   'initFrame'  method \n");
        [super initFrame:frameRect];            //   initialize SimpleView object
        primitiveToDisplay = CIRCLE;            //  CIRCLE primitive displayed initially
        fillFlag = 0;                           //  no fill initially
        sliderYvalue = 0.0;                     //   primitive at the bottom of SimpleView initially
        return self;
}
```

SimpleView.m (continued)

```
                                //  displayCircle is invoked when the “Circle” radio button is clicked
- displayCircle:sender          //  We set up a target/action connection with this method in IB.
{                               //  IB generated the skeleton, the “guts” were inserted.
        printf ("Executing  'displayCircle'  \n");      //  these 3 lines were inserted
        primitiveToDisplay = CIRCLE;
        [self display];         //  "display" message is sent to this SimpleView object
                                //   The "display" method sets up a drawing context
        return self;            //         and then invokes our "drawSelf" method.
}

- displaySquare:sender         //  very similar to the displayCircle method above
{                               //  invoked whenever the “Square” radio button is clicked
        printf ("Executing  'displaySquare'  \n");
        primitiveToDisplay = SQUARE;
        [self display];
        return self;
}
                                //  fillToggle is invoked whenever the “Fill” switch is clicked
- fillToggle:sender             //  We set up a target/action connection with this method in IB.
{                               //  IB generated the skeleton, the "guts" were inserted.
        printf ("Executing  'fillToggle'  \n");
        fillFlag = !fillFlag;                   //  toggle the fill flag (inserted)
        [self display];
        return self;
}
                                //  sliderMoved is invoked whenever the Slider is moved
- sliderMoved:sender            //  We set up a target/action connection with this method in IB.
{                               //  IB generated the skeleton, the "guts" were inserted.
        printf ("Executing  'sliderMoved'  \n");
        sliderYvalue = [sender floatValue];     //  read slider value from slider object
        [self display];                         //     (the "sender" of the message
        return self;                            //      which invokes this method)
}
```

SimpleView.m (continued)

```
                // The drawSelf:: method is invoked when SimpleGraphics is launched.
                // It is also invoked whenever a "display" message is received by this object.
-drawSelf:(NXRect *)rects :(int) rectCount       // This entire method was inserted.
{
        printf ("Executing   'drawSelf' \n \n");
                                        //  The "PS" functions are Display PostScript functions.
                                        //  The NX functions are NeXTstep graphics functions.
        PSsetgray (NX_WHITE);           //  set the drawing color to white
        NXRectFill (&bounds);           //  fill the entire SimpleView with a white rectangle
                //  "bounds" is an inherited C struct  initialized to SimpleView's dimensions
        PSsetgray (NX_BLACK);           //  set the drawing color to black
        NXFrameRect(&bounds);           //  draw a black boundary around all of SimpleView

        PSnewpath();                    //  begin a new outline path
        PSsetlinewidth (3.0);

        switch  (primitiveToDisplay) {          //  display a circle or square depending
                                                //    on the value of "primitiveToDisplay"
                case CIRCLE:    PSarc (bounds.size.width/2.0 , sliderYvalue,
                                        40.0, 0.0, 360.0);      //  radius is 40.0
                                break;
                case SQUARE:    PSmoveto (bounds.size.width/2.0 - 40.0 , sliderYvalue - 40.0);
                                PSrlineto (0.0, 80.0);
                                PSrlineto (80.0, 0.0);          //  relative line draws
                                PSrlineto (0.0, -80.0);
                                PSclosepath();
                                break;
        }       /*  end switch  */

        if   (fillFlag)     PSfill();       //  "paint" the enclosed area on the screen
        else                PSstroke();     //  "paint" the outline path on the screen
   return self;
}   /* end drawSelf */
@end                    // end SimpleView implementation
```

Project 3 Wrap-up

We

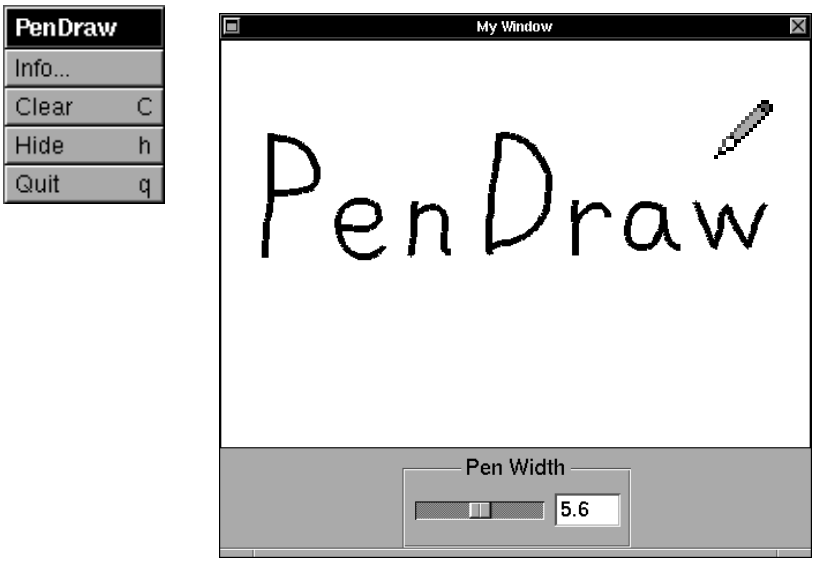
- created a subclass “SimpleView” of the “View” class
- customized and instantiated the subclass (to get an object of the “SimpleView” class)
- added the following control objects to the interface
  1. two radio buttons (in a cosmetic Box object)
  2. a “switch” button
  3. a slider
- added 4 methods to the “SimpleView” object (one for each of the control objects mentioned above)
- made target-action connections between these control objects and methods
- discovered how to display graphics in a custom View object by inserting Display PostScript and NeXT graphics functions in the View’s “drawSelf::” method (which is invoked whenever a “display” message is received).

Note: to get copies of these source files, see page 73

Project 4 - PenDraw

(allow a user to draw with a "pen" in a custom View object)

After a little "drawing", PenDraw's main menu and window will look like



The **main objectives** of PenDraw are to learn how to

- set up an event loop to handle mouseDown and mouseMoved events in a custom View
- change the cursor (to the "pencil" at the right)
- set up a correspondence between a Slider and Form.

Project 4 Step by Step

1. **Launch IB** and choose **New Application**, **Save As** and then **Project** from its File submenu. Save as "**proj4/PenDraw**".

2. **Create a subclass of the View class** called "**PenView**" by opening the Classes window, highlighting **"View**" in its browser, dragging to **Subclass** in the "Operations" pull-down list and renaming "MyView".

3. **Add a custom "View" object of the "PenView" class** by performing steps 4-5 below.

4. Drag a **"CustomView" icon** from the Basic Views palette into "My Window".

5. **Change the class of the "CustomView" icon (object) to "PenView"** by dragging to **Attributes** in the Inspector's pop-up list and then double-clicking on **"PenView"**.

6. Make "PenView" as large as "My Window" (by dragging on its handles) except leave about an inch at the bottom for a Box, Slider and Form.

7. **Add a Box object** as in the screen dump on page 62. Rename the Box object "Pen Width" and make the text larger.

8. **Add a Slider object** as in the screen dump on page 62. Change its range to 1-10 and initialize it to 1 in the Slider Inspector.

9. **Add a Form object** inside the Box to the right of the Slider. Alt-drag on the bottom middle handle to get rid of one of the Forms and then delete the "Field 1:" text on the remaining Form (double-click and use the Delete key).

   (We use a Form object here because the Text object can't handle floating point data.)

10. **Add a menu item called "Clear"** below "Info" in PenDraw's main menu (drag "Item" from Menus palette and rename). Delete the "Edit" submenu.

11. **Add a key-equivalent to the "Clear" menu item** by double-clicking on the right side of the menu item and typing an upper-case "C".

12. **Add an outlet called "formOutlet" to the PenView class.** (Click on PenView and then drag to Class in the CustomView Inspector.)

13. **Add actions methods called "changePenWidth" and "clearView" to the PenView class.**

14. **Set up an "outlet" connection** from **"PenView" to the Form.** Double-click on **"formOutlet:"** in the CustomView Inspector. (Make sure the **gray**, not black, rectangle surrounds the Form.)

15. **Set up an "action" connection from the Slider to "PenView".** Double-click on the **"changePenWidth:"** method in the Slider Inspector.

16. **Set up an "action" connection from the "Clear" menu item to "PenView".** Double-click on the **"clearView:"** method in the MenuCell Inspector.

17. **Add the "pencil.tiff" file to the project** by highlighting "**.tiff**" in the Files Project Inspector, clicking the "**Add**..." button, and finding the file in the / **NextDeveloper**/**Examples**/**Draw** folder. Copy the "pencil.tiff" file into the Project folder when prompted.

18. ***Create class interface** ("PenView.h") and **class implementation** ("PenView.m") **files** by dragging to **Unparse** in the "Operations" pull-down list in the Classes window. Add these two files to the project.

19. **Insert the code** on pages 65-69 **into the PenView.h and Penview.m files** (or copy these files into the proj4 folder - see the note below).

20. **"Make" PenDraw** using IB's File submenu item or in a Shell window.

21. **Launch PenDraw** and test its "PenWidth" Slider, "Clear" menu item and key-equivalent 'C'. Note how the cursor changes within the PenView border.

***Note:** **You can skip steps 18 and 19 above** if you have the PenView source files available. In this case, add these files to the Project using the Files Project Inspector in a similar fashion to what was done with the ".tiff" file in step 17 above. This time highlight "**[.hm]**".

I will **e-mail** you copies of the source code for projects 3 and 4 in this tutorial if you send a request to the address **mahoney@csulb.edu**

PenDraw Objective-C Files

PenDraw_main.m

This file is generated by Interface Builder when "Project" is chosen in IB's File submenu. It is exactly the same as the SimpleGraphics_main.m file on page 58, except that the PenDraw.nib interface section is loaded instead of the SimpleGraphics.nib section.

PenView.h

```
// Generated mostly by Interface Builder when the "Unparse" command was chosen,
// The last 3 instance variables and the last 4 (OVERRIDDEN) methods were inserted.
                                                        //      Mike Mahoney,  2/1/91.
#import <appkit/View.h>
@interface PenView:View
{
        id formOutlet;          //  pointer to the Form object
        float penWidth;         //  contains the width of the pen (initialized in initFrame)
        id pencilCursor;        //  pointer to NXCursor object which looks like a pencil
        NXPoint hotSpot;        //  point on the pencilCursor image that's aligned
}                               //      with the mouse (initialized in initFrame)
//  OUTLET INITIALIZATION METHOD
//      The system will initialize IB-set outlets. However, if we want to perform other
//       initializations we can use "set" methods. They are automatically called at launch time.
- setFormOutlet:sender;         //  sets initial float value in Form to Slider value

//  TARGET/ACTION METHODS
- changePenWidth:sender;       //  invoked when the "penWidth" Slider is moved
- clearView:sender;            //  invoked when the "Clear" menu item is chosen

//   OVERRIDDEN VIEW METHODS              //  these were inserted
- initFrame:(const NXRect *) frameRect;
- drawSelf:(NXRect *)rects :(int) rectCount;     //   used only to clear PenView
- mouseDown: (NXEvent *) ptr;           //   to grab control when mouse pressed in PenView
- resetCursorRects;                     //   resets cursor rectangle (automatically
@end                                    //      invoked, don't invoke directly
```

PenView.m

```
//  PenView.m
//  Skeleton generated by Interface Builder.
//  The rest was inserted by Mike Mahoney on 2/1/91.

#import "PenView.h"

#import <appkit/Application.h>          // these six imports were inserted
#import <appkit/NXCursor.h>
#import <appkit/Form.h>
#import <appkit/Slider.h>
#import <appkit/Window.h>
#import <dpsclient/wraps.h>


@implementation PenView

- initFrame:(const NXRect *) frameRect
{
        [super initFrame:frameRect];

        penWidth = 1.0;
                                                        //  load pencilCursor
        pencilCursor = [NXCursor newFromImage:[NXImage newFromSection:"pencil.tiff"]];

        hotSpot.x = 0.0;    hotSpot.y = 15.0;     //  hotSpot is the point on the  16 x 16  cursor
        [pencilCursor setHotSpot:&hotSpot]; //    image that's aligned with the mouse
                                                //   the origin of a cursor is at upper left corner
        return self;
}
```

PenView.m (continued)

```
                                    //   This method was included to initialize the Form text to the
- setFormOutlet:sender              //       penWidth value set in initFrame above.
{                                   //  It also initializes the formOutlet, but that would have been
        formOutlet = sender;        //       done automatically by the system if this wasn't here.
        [formOutlet setFloatValue: penWidth at:0];
        return self;
}


- changePenWidth:sender   // gets the penWidth value from the Slider object (the "sender")
{
        penWidth = [sender floatValue];                 // get the penWidth value
        [formOutlet setFloatValue: penWidth at:0];    //  update Form with new penWidth value
        return self;
}


- clearView:sender                  //   used only to clear PenView
{                                   //   see the drawSelf method below
        [self display];            //  sets up a graphics context and invokes drawSelf below
        return self;
}


- drawSelf:(NXRect *)rects :(int) rectCount;            //   used only to clear PenView
{                                                      //   see the clearView method above
        PSsetgray (NX_WHITE);
        NXRectFill (&bounds);                  //  fill entire PenView with white
        PSsetgray (NX_BLACK);
        NXFrameRect (&bounds);                 //  draw a black frame around PenView

        return self;
}
```

PenView.m (continued)

```
- resetCursorRects               //  resets PenView's cursor rectangle
{                                //    so pencil shows up inside PenView
        [self addCursorRect:&bounds cursor:pencilCursor];     //  set cursor rectangle
        return self;                        // for details look up these methods in View class docs.
}


- mouseDown: (NXEvent *) theEvent   //  our own implementation of mouseDown
{       //   invoked every time the mouse button is pressed down inside PenView

        NXPoint currentPos, oldPos;
        NXEvent *nextEvent;
        BOOL   looping = 1;     // true
        int             oldMask;                //  mouse event masks
        int             checkMask;


        oldMask = [window eventMask];

        checkMask = NX_MOUSEUPMASK | NX_MOUSEDRAGGEDMASK;

        [window setEventMask: (oldMask | checkMask)];

        [self lockFocus];       //  locks the PS focus on PenView prior to any drawing
                                //  lockFocus/unlockFocus not needed in drawSelf
        oldPos = theEvent -> location;          //  get starting location for a very short line
        [self convertPoint: &oldPos fromView:nil];      //  converts oldPos from "My Window"
                                                        //      coordinates to penView's coords.
        PSsetlinewidth (penWidth);
```

PenView.m (continued)

```
        while (looping)  {        //  takes over event loop while mouse is pressed down
                nextEvent = [NXApp getNextEvent: checkMask];
                                                        //  if mouseup, looping = FALSE
                looping = (nextEvent -> type != NX_MOUSEUP);

                if (looping)  {
                        currentPos = nextEvent -> location;
                        [self convertPoint: &currentPos fromView:nil];

                        if ((currentPos.x != oldPos.x) | (currentPos.y != oldPos.y))  {
                                PSnewpath ();                   //  don't use drawSelf because
                                PSmoveto (oldPos.x, oldPos.y); //    of  dynamic drawing
                                PSlineto (currentPos.x, currentPos.y);
                                PSstroke ();
                                oldPos = currentPos;
                                [window flushWindow];           // needed else drawing is only
                                                                // shown when Slider is moved
                        }
                }
        }   /* end while */

        [self unlockFocus];             //   companion to [self lockFocus]

        [window setEventMask: oldMask];

        return self;

}    /* end of mouseDown */


@end    // implementation
```

Project 5 - Parse Example

(use source files from Project 2, KmToMi, to show how to use the Parse operation)

In this example we parse a class interface (.h) file so that the outlets and methods of the associated class are known to IB. The resulting app, called ParseKmToMi, will function exactly the same as KmToMi when complete. See the Project 2 screen dump on page 46 for a picture of the finished product.

The **Parse button** in the Operations pull-down list in the Classes window reads the class definition for the selected class from an interface file. IB looks for the class interface file in the current project folder. If IB finds it, then the class is displayed in its proper place in the Classes window browser.

You can only use the Parse button if you have previously specified the class's position in the class hierarchy and copied the files into the project folder.

A simpler method of adding an existing class to the hierarchy in IB is to drag the ".h" icon for the interface file into the File or Classes window. IB will then automatically parse the interface file.

Project 5 Step by Step

1. **Before launching IB,** create a folder called **proj5** and **copy the files ConvertObject.m and ConvertObject.h** from the **proj2** folder into it.

2. **Launch IB** and choose **New Application**, **Save As** and then **Project** from its File submenu. Save as "**ParseKmToMi**" in the **proj5** folder.

3. **Create a subclass of the Object class** called "**ConvertObject**" by opening the Classes window, highlighting **"Object**" in its browser, dragging to **Subclass** in the "Operations" pull-down list and renaming "MyObject".

4. **Parse the "ConvertObject.h" file** by dragging to **Parse** in the **Operations** pull-down list. Click "**Replace**" when you get the panel which says "Parsed file has different actions or outlets".

   Drag to **Attributes** in the Inspector's pop-up-list and note that the outlets "kmOutlet" and "miOutlet" and the "convertMethod" method appear (since they were specified in the "ConvertObject.h" file).

5. **Create an instance object (and icon) of the ConvertObject class** by dragging to **Instantiate** in the "Operations" pull-down list.

6. **Add the ConvertObject class to the project** by highlighting "**[.hm] (class)**" in the Files Project Inspector, clicking "**Add**..." and double-clicking **ConvertObject.m** in the Open panel.

7. **Build the KmToMi interface** and **make the KmToMi connections** by working through the appropriate steps in Project 2. You will need to do the following steps: 3-15, 25-26, and 30-32 (inclusive). Then "make", launch and test ParseKmToMi.

Note: This project could have been done slightly more easily by deleting steps 1,3 and 4 and performing the following after step 2.

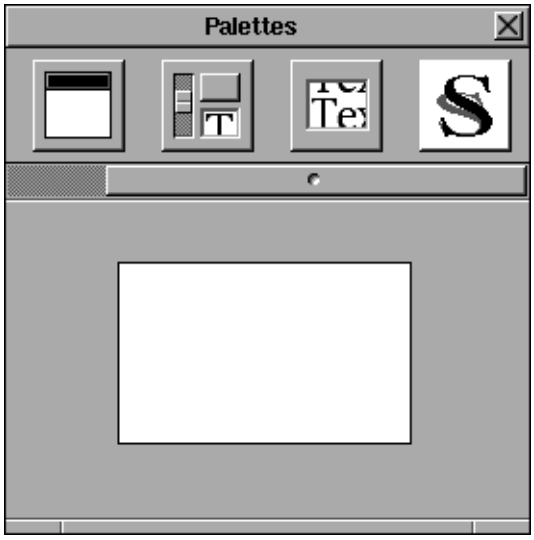
Locate the ConvertObject.h file in the File Viewer (in the proj2 folder) and drag it's icon into IB's File window.

The "Classes" suitcase icon opens and the ConvertObject class is added to the class hierarchy for the project. Double-click on the "Classes" suitcase icon to see the ConvertObject class in the hierarchy.

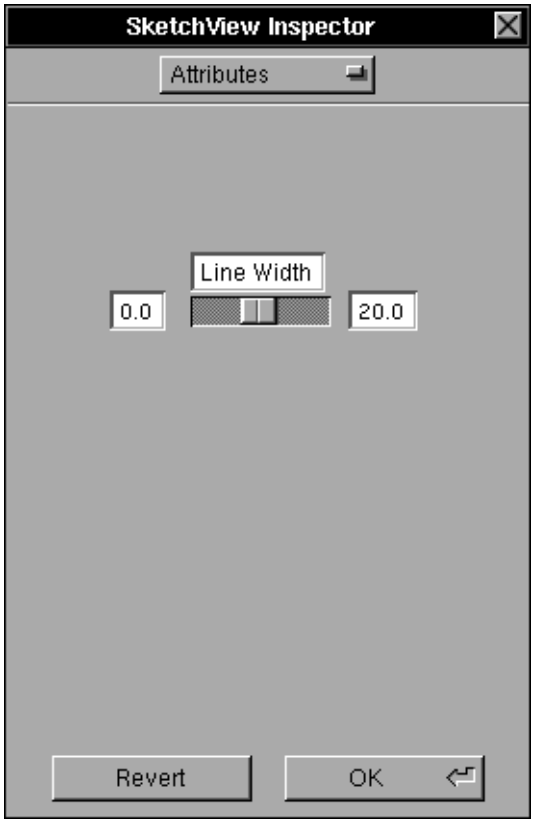
Project 6 - SketchPalette

(add a user-defined Palette to IB's Palettes window)

After setting up and loading the new Palette, IB's Palettes and Inspector windows will look like:



Note the new Palette and the new slider in the Palettes window.



This example can be found on-line by searching for **SketchPalette** in the **"Developer RelNotes**" documentation using **Digital Librarian**. This example is on-line only in the extended software release.

←The main window of an application which uses SketchPalette with a line width of about 6.

What to do next

1. Projects in this tutorial:

Carefully work through the projects in this tutorial on a computer running NeXTstep 2.0 and think about objects and connections as you do them.

I will **e-mail** you copies of the source code for projects 3 and 4 in this tutorial if you send a request to the address **mahoney@csulb.edu**

2. NeXT Interface Builder Tutorial and Reference:

Carefully read through the **Chapter 8 of the *NeXTstep Concepts* manual.** In particular, work through **projects** in the Interface Builder tutorial there.

3. *The NeXTstep Advantage* booklet

This 107 page booklet from NeXT gives an excellent overview of the NeXTstep development environment and guides the reader through the development of a nontrivial example called Plotter. The Plotter app contains many of the features found in NeXTstep applications, including Listener/Speaker and delegates which weren't discussed in this tutorial.

4. Examples (including source code) in the /NextDeveloper/Examples folder.

Many valuable features and techniques are demonstrated in these examples. Looking through them is probably the fastest way to learn how to add features to your own applications using Interface Builder, the Objective-C Language and PostScript programming. To get an overview of each of the examples, read the contents of the **README.rtf** file which resides in the **/NextDeveloper/Examples folder.**

Brief Glossary

(mainly Objective-C Language terms)

object - a programming unit that groups together a data structure (instance variables) and the operations (methods) that can use or affect that data; the central focus of object-oriented programming.

class - a particular kind of object. Objects that have access to the same methods and have the same types of instance variables belong to the same class. A class definition declares the instance variables and defines the methods for all members of the class.

subclass - for any given class of objects, any class that's one step below it in the inheritance hierarchy.

inheritance hierarchy - the hierarchy of classes that's defined by the arrangement of superclasses and subclasses. Every class (except Object, which is at the root of the hierarchy) has a superclass, and any class may have an unlimited number of subclasses. Through its superclass, each class inherits from those above it in the hierarchy.

method - a procedure that can be executed by an object

message - the method name and arguments that are sent to an object; it tells the receiving object what to do. The format is [receiver methodname (plus arguments)].

action message - a message sent by a control object (such as a Button or Slider) in response to a user action (such as clicking the button or dragging the slider's knob).

instance variable - a variable that's part of an object's private data structure. Instance variables are declared in a class definition and become part of all the objects that are instances of the class.

target - the object that receives action messages from a control (button, menu item, etc.)

control object - a graphical object (e.g. button, slider) which a user can operate to give instructions to an application

outlet - a pointer to an object which needs to be sent messages

interface file - a file that declares the interface to a new class

implementation file - a file that contains the code that implements a new class

References

The most important reference is the ***NeXT Developer's Library***, which consists of 10 manuals. One of these manuals, *NeXTstep Concepts*, is the best place for beginning developers. Unfortunately, the version of *NeXTstep Concepts* shipped in April 1991 is based on NeXTstep 1.0, and it is not on-line with the 2.0 Release. For further references, see the **Suggested Reading** in the **/NextLibrary/Documentation/NextDev/Summaries/Reading.rtf** file.

- **NeXTstep User Interface Guidelines**: Chapter 2 of *NeXTstep Concepts*, the files in the **/NextLibrary/Documentation/NextDev/Notes/UIUpdate.rtf** folder and the **BusyBox example.**

- **Interface Builder**: Chapter 8 of *NeXTstep Concepts* and the file **InterfaceBuilder.rtf** in the **/NextLibrary/Documentation/NextDev/ReleaseNotes** folder**.**

- **Objective-C and Object-Oriented Programming**: Chapter 3 of *NeXTstep Concepts*; *Object-Oriented Programming: An Evolutionary Approach* by Brad J. Cox (Addison Wesley, 1986); *Intro to Objective-C on the NeXT Machine* by Gerrit Huizenga (obtain by downloading the file **/pub/next/docs/ObjC-frame.ps.Z** from the **sonata** site given below).

- **PostScript and Drawing:** Chapter 4 of *NeXTstep Concepts; PostScript Language Tutorial and Cookbook* and the *PostScript Language Reference Manual, 2nd edition,* Adobe Systems, (Addison Wesley, 1985, 1990).

- **Internet archive sites:** **lynx.cs.orst.edu**
  nova.cc.purdue.edu
  sonata.cc.purdue.edu
  wuarchive.wustl.edu

Acknowledgments

Congratulations to **Jean-Marie Hullot** (and **Lee Boynton**) for a fantastic tool that will save many programmers many hours and allow nonprogrammers to easily design user interfaces for NeXTstep apps.

Thanks to **Bruce Blumberg, Conrad Geiger, Eric Larson**, **Ali Ozer**, **Sara Benson** and of course **Steve Jobs** of **NeXT Computer, Inc.**

Thanks also to **Dennis Volper, Dave Bradley, Chuck Schneebeck, Bob Clover, Lorraine Rapp, Yvette Perry, Stein Tumert** and especially **Henry Chiu** at **CSU Long Beach**.

A special thanks to the 1990 and 1991 CHI tutorials chairs, **Wendy Kellogg** (IBM) and **Tom Hewett** (Drexel University).

These notes were composed using **FrameMaker**™ on a NeXT Computer.

All of the windows and screen dumps were captured with the **Grab** Application and imported (thanks **Keith Ohlfs** of NeXT - for **Icon** too).

TradeMarks

NeXT, **Workspace Manager, Application Kit, Interface Builder**, and **Digital Librarian** are trademarks, and the **NeXT logo** and **NeXTstep** are registered trademarks of **NeXT Computer, Inc.**

WriteNow is a registered trademark of **T/Maker Company.**

PostScript and **Display PostScript** are registered trademarks of **Adobe Systems Incorporated**.

UNIX is a registered trademark of **UNIX Systems Laboratories**.

FrameMaker is a registered trademark of **Frame Technology Corporation.**