The best way to learn about NeXTSTEP's benefits is to follow the development process from start to finish. Here, you'll see how NeXTSTEP helps you create programs that closely match user needs, using a minimum of unique code. You'll also see how NeXTSTEP simplifies each step of development from design to coding to debugging.

Now that you've had a chance to examine the foundations of NeXTSTEP, it's time to see how you go about building applications on them.

YourCall is an application for use by customer support representatives to log calls received on a toll-free hotline. It's the kind of application that any company providing telephone support might need—albeit a very simple version of one.

As with all NeXTSTEP applications, many of the components needed to implement the YourCall application are provided by the Application Kit. In the steps that follow, you'll see how the basic architecture of YourCall is provided by NeXTSTEP. You'll also see how many features specific to this application—including its windows, its other user interface components, its printing and faxing capabilities—are added with no programming effort.

A few of the objects used to implement YourCall will be designed specifically for this program. The following steps show that NeXTSTEP provides well-defined ways to integrate such custom objects with those provided for you. You'll see how the NeXTSTEP programming model simplifies coding these objects and connecting them with the rest of your application.

STEP 1: DESIGNING THE APPLICATION

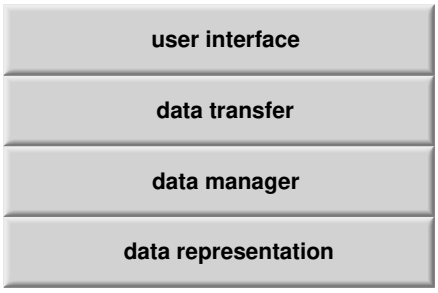
The object-oriented approach simplifies both design and programming. Design is simpler because objects map naturally onto the structure of a programming task. What's more, modular design lets you begin programming without specifying every single procedure and data structure. NeXTSTEP applications can be implemented by progressive refinement—you can work out many design details as they're needed.

The Request

The Customer Support Department wants a simple application—YourCall—to store customer calls. For every customer phone call received, the department wants to keep some basic information: customer name, address, and phone, as well as the customer's question and the answer. In addition, support representatives want to use this application to retrieve calls by name. They want to perform all these operations from a single, simple form.

Program Structure the Traditional Way

As with most application design tasks, the functional requirements of the YourCall application suggest a program structure:



At the lowest level, the specification from Customer Support suggests a *data representation*, a set of containers or records to capture the data for each call. These representations include fields for each data component of a particular call, such as customer, question, and answer.

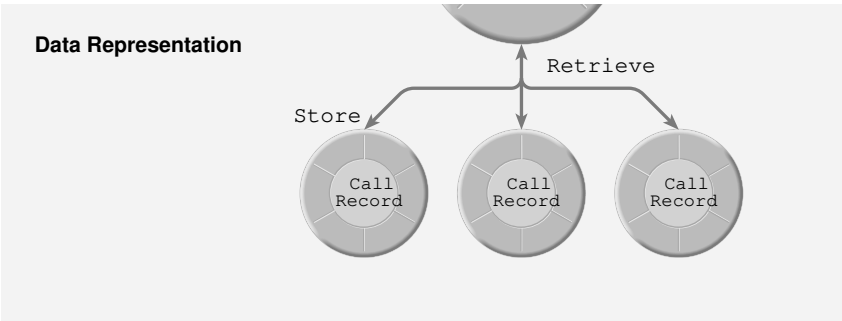
To handle storage and access to these records, the application needs a *data manager*. This manager should be able to store and retrieve records in a file, and look up records by a key value such as customer name.

To move data between the user interface and the data management layers, the application needs a *data transfer* or control layer. This layer interprets actions in the user interface—such as pushing a "Save Call" button—and translates that action into a request to the data manager.

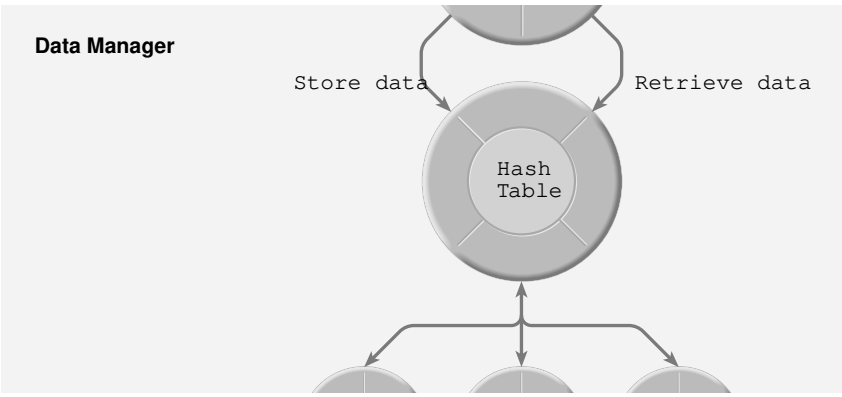
Finally, the application needs an easy, intuitive *user interface*, a simple form for input and display of call information. This user interface should allow a support representative to enter and edit the data for a call. The same interface should be able to retrieve and display call information by customer name. It should also offer an easy and consistent way for users to initiate these save and retrieve actions—for example, by clicking buttons with a mouse.

Program Structure the Object-Oriented Way

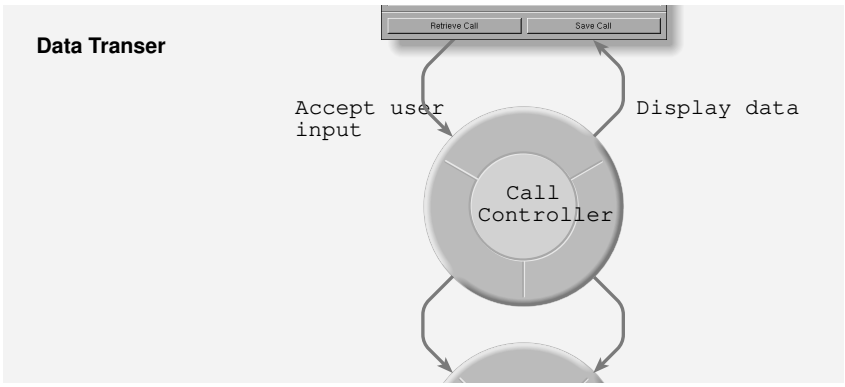
Through object-oriented design, the specification from Customer Support is easily captured as distinct objects. Most of these objects are provided by NeXTSTEP; just two unique object types need to be programmed to implement objects specific to YourCall.



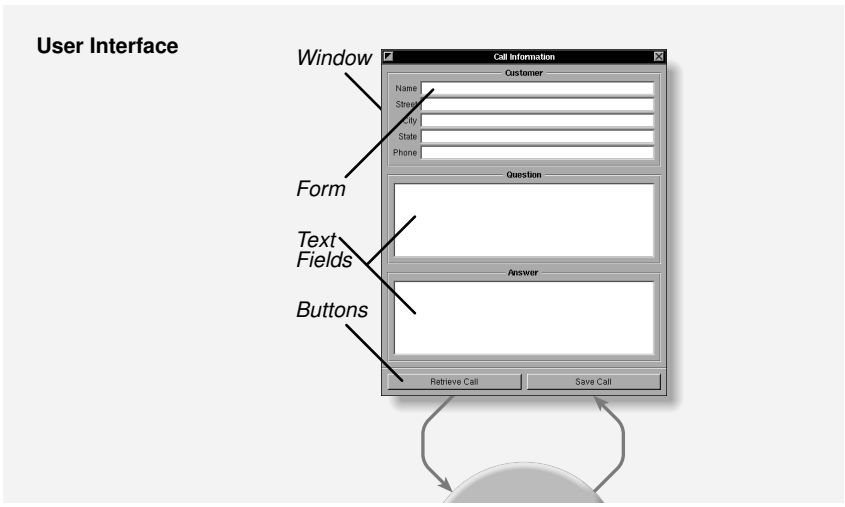
The basic unit of information that Customer Support wants to capture is a call. This suggests that each call could be represented by an object—a CallRecord. These objects are specific to this application, so CallRecord is one of the custom classes needed by YourCall. (In this example, as in all of NeXTSTEP, class names are capitalized.)



To manage CallRecords, YourCall needs an object that can store and retrieve objects by a specific data value—the customer name. HashTable is a basic storage class provided by NeXTSTEP. A HashTable object can store and retrieve other objects by key value. Taking advantage of an Objective C feature known as *archiving,* a HashTable object can also save and retrieve objects on disk.

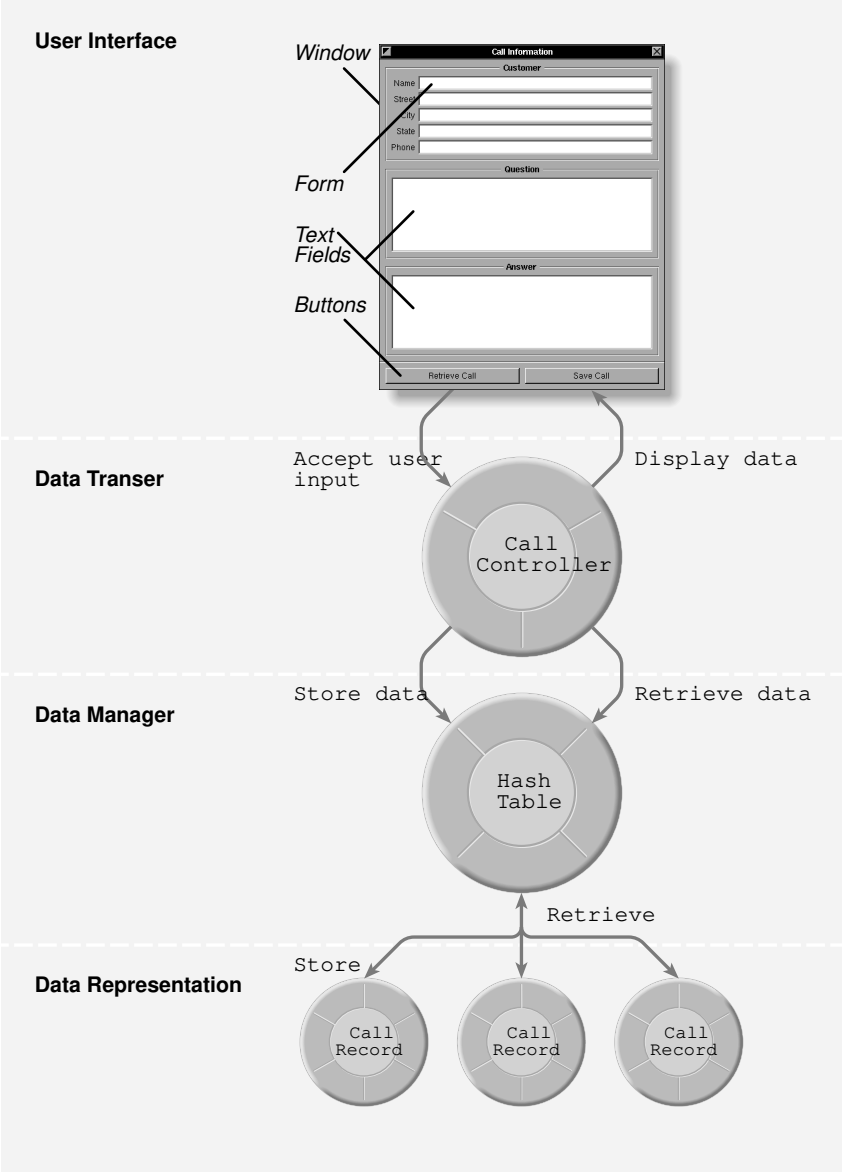


To transfer data between the user interface and the data manager, YourCall uses a CallController object. This object needs to interact with a variety of objects, including the TextFields that display data and the Buttons that accept user input, the HashTable that manages the data, and, through the HashTable, the individual CallRecords. Because this interaction is unique to YourCall, CallController is the second custom class defined for this application.



Finally, while the user interface for YourCall is unique, the objects that make up the interface aren't. Like nearly all NeXTSTEP user interfaces, YourCall's on-screen display can be assembled entirely from objects provided by the Application Kit through Interface Builder. The main component is a Window, containing a Form for the customer information, TextFields for the question and answer, and Buttons that the user clicks to signal to the CallController to retrieve and save calls. In addition to defining the basic appearance of these objects, the Application Kit defines standard ways for them to interact with other objects in an application.

With all these objects described, the picture of the YourCall application is complete:



For every application, including YourCall, NeXTSTEP provides another fundamental object: the Application object. Like the shaded background of the illustration, the Application object provides a context for the rest of the objects in the application. It establishes a connection with the windowing system and makes sure all objects required by the application are ready when the application starts. As the application runs, the Application object distributes *events* (mouse clicks and keystrokes) to the appropriate objects in the user interface. The Application object is discussed in detail in "The NeXTSTEP Application Framework" later in this guide.

Designing the Custom Objects

The next phase of application design is to specify the methods that the custom objects will use to interact with other parts of the program. This phase doesn't involve any programming—it's simply specifying the names of methods that

you will implement later. Once these methods are specified, object-oriented modularity allows the project manager to divide responsibility for YourCall among the programming team: one programmer could implement CallController, while another implements CallRecord and a third works on the user interface.

For a CallController to intercept user actions and turn them into requests to the data manager, it must have methods to respond to the two Button objects in the user interface. The method retrieveCall: will look at the name in the form's Name field, request a CallRecord from the data manager for that name, then display that record in the form. saveCall: will create a new CallRecord for the information in the form and ask the data manager to save it. (Note that method names are indicated in bold type, and that the colons are part of the method names—you'll see how these colons are used when you get to the actual coding for YourCall.)

A CallRecord contains the fields for a particular call: name, address, phone, question, and answer. The internal storage for these fields won't be visible to the rest of the program. Instead, the methods for setting and retrieving the data will be. For example, the method setName: will set the customer name to a particular character string, while the method name will return the name string. The other methods for setting and retrieving data will have similar names. For convenience, the data accepted or returned by all of these methods will be character strings.

Designing the Application: Summary

In this step, you've seen how the object-oriented approach maps onto a programming task, enabling you to specify objects at a high level, then divide them among a programming team for implementation. Once a program's objects, their behavior, and the interfaces for invoking that behavior are specified, you can turn to project development.

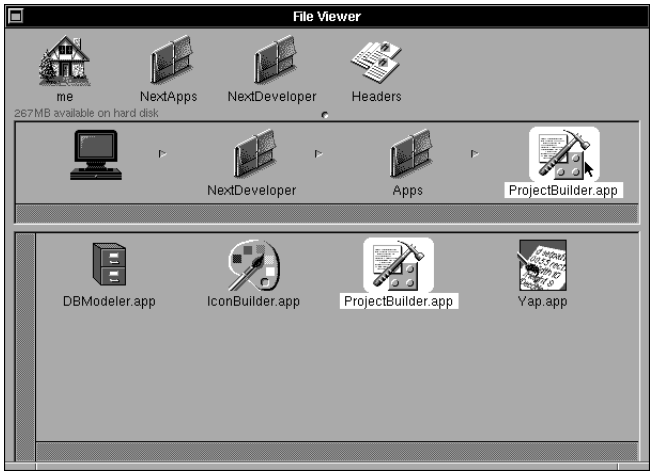
STEP 2: CREATING THE PROJECT

In NeXTSTEP, hands-on application development begins with Project Builder. This NeXTSTEP application not only manages project development, it also provides a work center for accessing other

application development tools. And it demonstrates some of the powers of NeXTSTEP available to all applications.

The Workspace Manager‰—which is itself a NeXTSTEP application—provides a simple way to launch all applications. Applications are represented in Workspace Manager by icons. To start an application, you simply use the mouse to double-click an application icon.

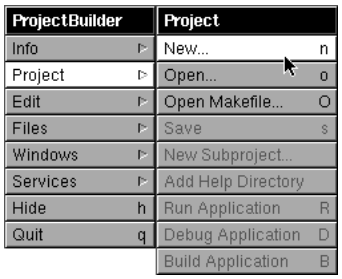
Project Builder is located in the directory path /NextDeveloper/Apps. To start Project Builder, you locate its icon in the Workspace File Viewer, then:



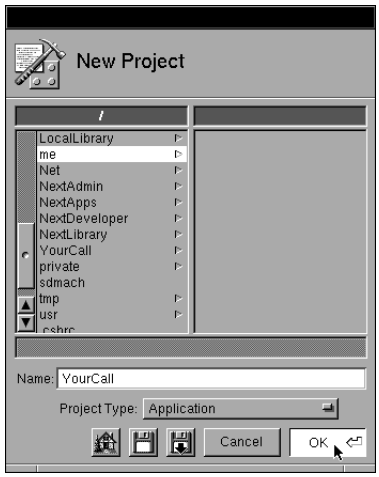
Point to the Project Builder icon and double click

The Project Builder menu appears

To begin a new project:

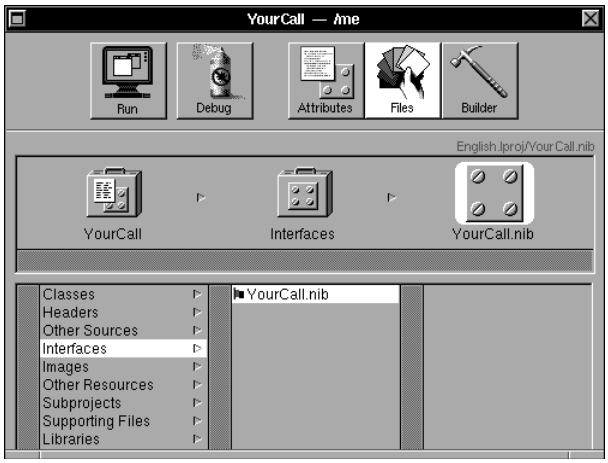


Point to Project, press the mouse button, and choose New from the Project menu



Click in the Name field, type the name “YourCall,” then click OK

When you click OK, Project Builder creates the new project.



A project window appears with the title "YourCall." When it first appears, this window shows the Files display, the display you use to manage your application's component files. The file YourCall.nib displayed there is a user interface template—in the next step you'll see how to open this file in Interface Builder and create an interface from it. Browsing through this display, you can see other files created for the project by Project Builder, such as YourCall_main.m (the standard main file required by all C applications), and a Makefile for building the application. In addition to these files, Project Builder creates the Project folder, a directory to store all source files and project management files. The Project folder contains files listed in the Files display, as well as the project management file used by Project Builder itself, PB.project.

Creating the Project: Summary

Project Builder provides the starting point for all NeXTSTEP applications. When you start a project with Project Builder, it automatically creates a project directory and standard source files for the application. As you add new source files, Project Builder tracks them for you. In the next step, you'll see how to use one of the standard source files to begin developing the user interface.

STEP 3: CREATING THE USER INTERFACE

Interface Builder is a tool for creating user interfaces, creating and editing objects, and making the connections among objects that enable an object-

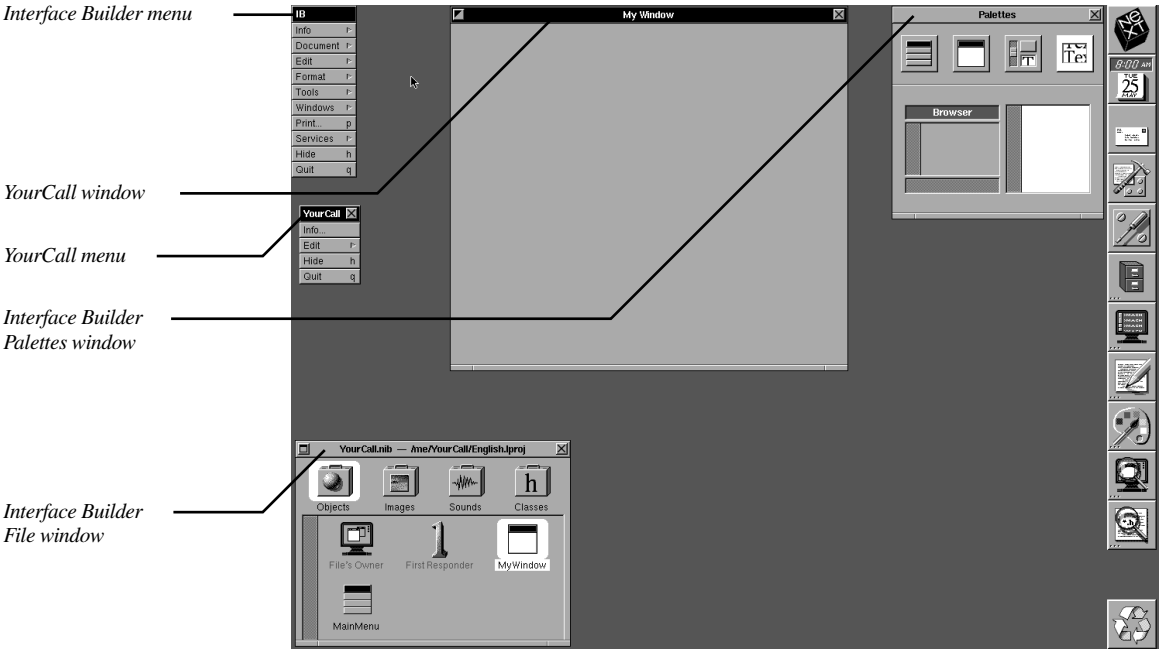
oriented application to work. In this step, you'll see how to use Interface Builder for user interface design, and get a glimpse at how to connect objects in an application.

Interface Builder takes full advantage of the object-oriented NeXTSTEP environment. As you build a user interface, you work with objects that will be used by the program. Interface Builder creates these objects for you, communicates with them to set their values, and stores them so that the application can retrieve and use them when it starts.

Because it works with "live" objects, Interface Builder lets you test the user interface without writing a single line of code. You can even give it to a potential user for trial and comment. You'll see how this works once the call form for YourCall has been created in Interface Builder.

Opening the Interface Builder File

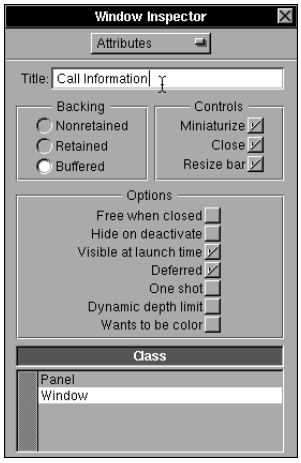
To start Interface Builder, double-click YourCall.nib in Project Builder. This template file, created by Project Builder, is actually an Interface Builder file. When you double-click the file, Interface Builder launches automatically—demonstrating how one NeXTSTEP application can communicate with another to provide cooperative services. When Interface Builder starts, you can see that the file provides Window and Menu objects. Project Builder automatically includes these important objects in the user interface file for every application.



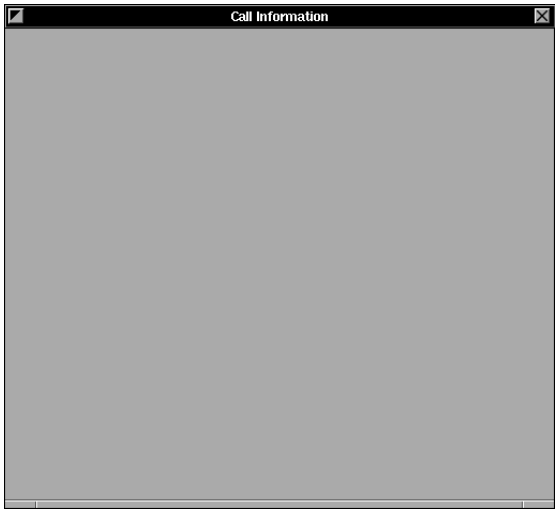
Setting Object Attributes

To let you set the attributes of user interface objects and connect them to other objects, Interface Builder provides an Inspector panel. It is one of the main ways Interface Builder lets you customize user interface objects provided by the Application Kit. The Inspector is provided by Interface Builder's Tools menu. One use of the Inspector is to rename objects. To rename the YourCall Window, make sure it's selected as above, then:

Choose Inspector from the Interface Builder Tools menu



Type "Call Information" in the Title field



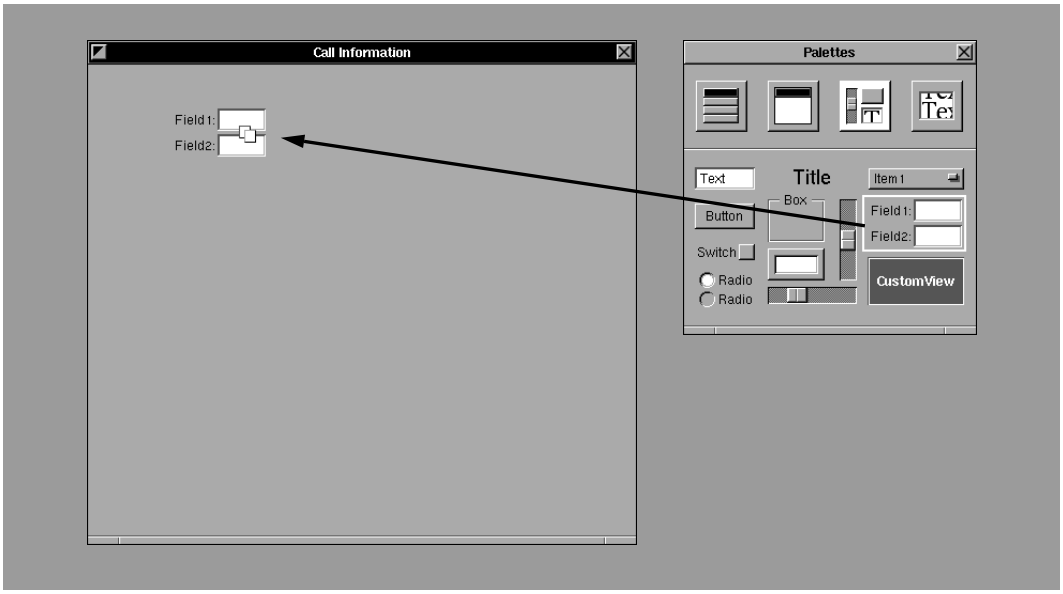
Press Return and the Window name changes

Adding Objects to the Interface

Another standard feature of NeXTSTEP is support for mouse-controlled drag-and-drop operations within and between applications. For example, a graphic may be added to a document by pointing to its file icon, pressing the mouse button, dragging the icon into the document, and releasing.

Interface Builder takes advantage of this capability to let you assemble a user interface from various objects. Interface Builder's Palettes window provides the user interface objects you use to compose your interface: Buttons, TextFields, Boxes, Sliders, and more. The following steps demonstrate how you use the Palettes window to drag and drop these components in the user interface.

YourCall uses a Form for customer information. To add the Form to the Call Information Window:



Point to a Form in the palette, press the mouse button, drag the Form to the Window, then release

To resize the Form:

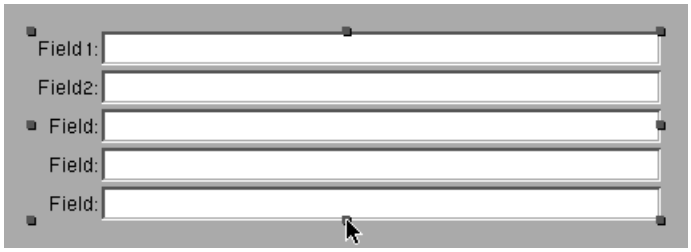
Grab the side handle

Drag it

Release

Then, to make five fields in the Form:

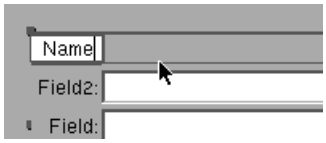
Hold the Alternate key and grab the bottom handle



Drag and release

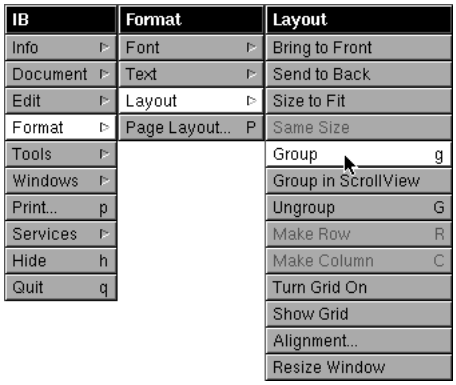
To label a field:

Double-click the field, then double-click its title

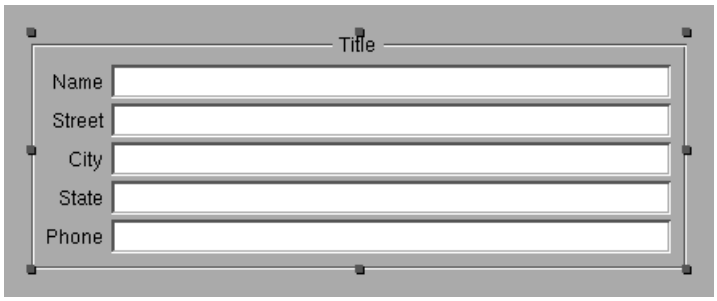


Type the label and press Return

Once the Form has been properly laid out, you can box and label it for easier identification. To do so, select the entire Form (not just a single field), then:



Choose Group from the Layout submenu



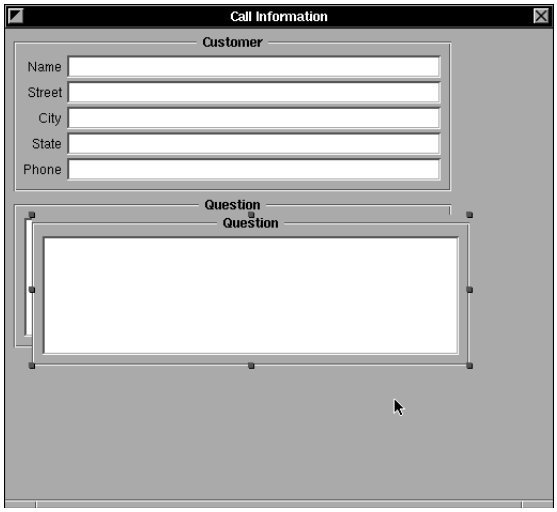
The Form is grouped

After editing the title above the grouping box to read “Customer,” you can proceed with adding the TextFields.

YourCall’s interface needs two TextFields, one labeled “Question” and one labeled “Answer.” Each should be the same size, and each should be boxed in a group of its own. Only their titles are different. To add a Question field:

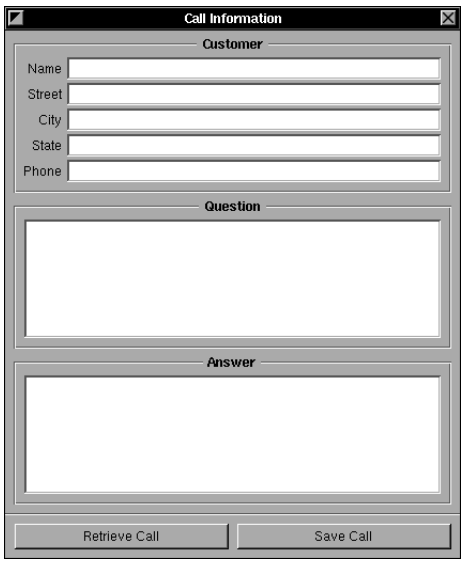
- Drag a TextField (labeled “Text”) from the palette to the interface.
- Choose Group from the Layout submenu to group the field.
- Title the field “Question.”
- Double-click the default entry “Text” and type Command-x to delete.

NeXTSTEP supports cut and paste operations on any selection: text, graphics, or other data. In Interface Builder, for example, you can cut and paste user interface objects. To create an Answer field, copy the Question field by selecting the grouped field and typing Command-c. Then type Command-v to paste the copied field in the interface

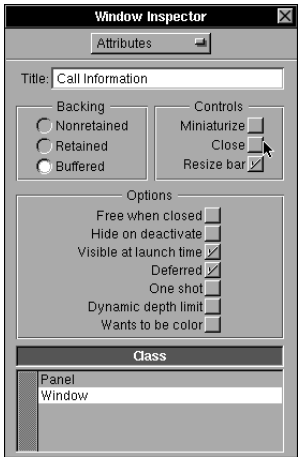


After repositioning and renaming the second field, you can finish the user interface. Drag a button from the palette to the window, Alternate-drag to

stretch it into two buttons, label each button, resize the window, and so on. When you're done, the user interface should look like this:



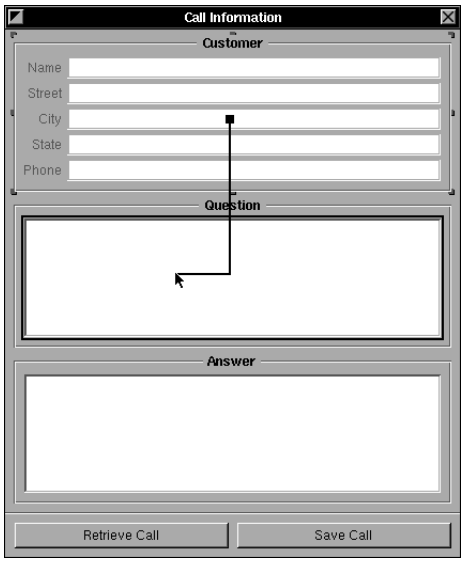
Finally, note that the Call Information window is the main component of a user interface. It should always be on the screen, and the user shouldn't resize it. You can control these aspects of Window behavior from the Inspector. So, with the Call Information window selected, click the Close and Resize buttons in the Inspector to turn them off.



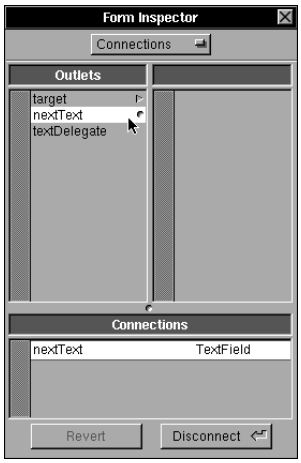
Making Connections in the Interface

In the Application Kit, there are many standard messages that objects can send to one another as an application runs. For example, both Form and TextField objects share a standard technique to implement tabbing between text fields using such messages. When the user presses the Tab key while typing in a Form or TextField, that object sends a message to another object—an *outlet* identified as nextText—telling the outlet object to select its text.

In step 5, you'll learn how outlets are implemented. For now, you'll see how to make the nextText connection with Interface Builder:



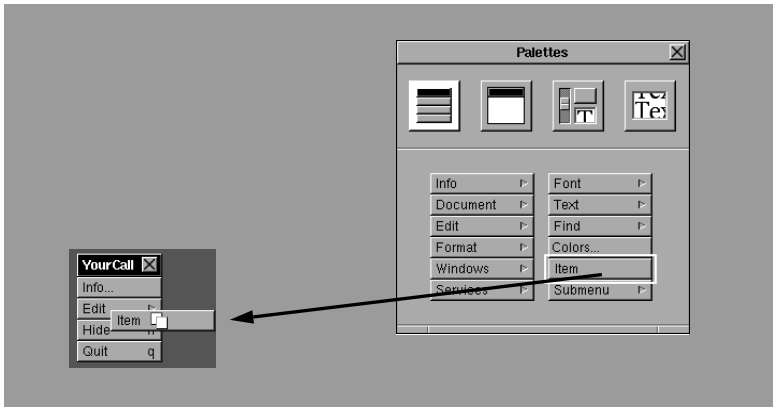
Choose the whole Customer Form, press Control, and drag to the Question TextField



Double-click the **nextText** outlet in the Inspector

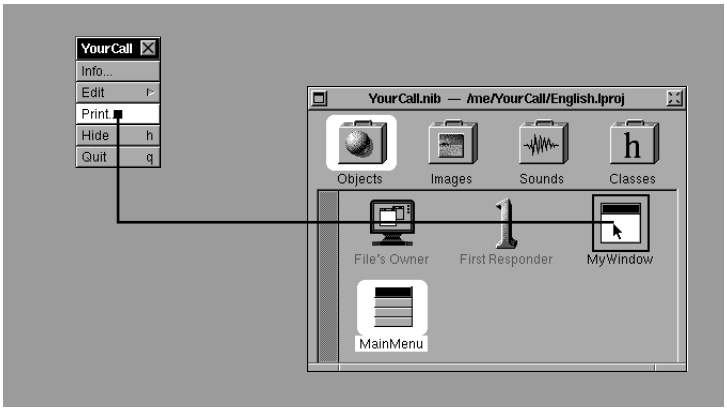
Similarly, make a nextText outlet connection from the Question TextField to the Answer TextField, and from the Answer TextField to the Customer Form. In Test mode, you'll see these connections at work.

Now, you can customize the menu for YourCall. The Menus palette comes with a variety of standard menu items, many of which implement behavior useful to any application. Other pre-defined features are easy to add by making simple connections. For example, to add printing:

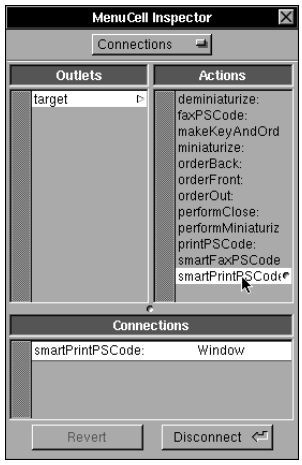


Choose the Menu palette, then drag “Item” to the menu

After editing the item to read “Print...” (the ellipsis indicates that a Print panel will appear when the user chooses this item):



Connect the item to the Window’s icon.

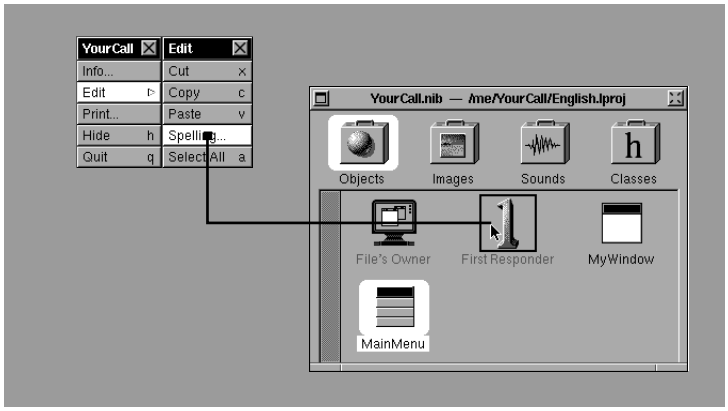


Double-click the **smartPrintPSCode:** action

Note in the inspector that the Window is the *target* of the Print MenuCell and that its *action* is smartPrintPSCode:. You’ll see in the next step exactly how targets and actions actually work.

Some operations aren’t associated with a specific object until the user actually uses the program. For example, spell checking should work on the text where the user is currently typing. The object-oriented feature *dynamic binding* allows an application to determine, as it runs, which object should receive a particular message. In Interface Builder, First Responder is a stand-in for whatever interface object a user has selected. For example, this would be the Name field when the user is typing the customer’s name, or the Question field when the user is typing the customer’s question.

To enable spell-checking, add an item to the Edit menu and label it “Spelling...,” then:

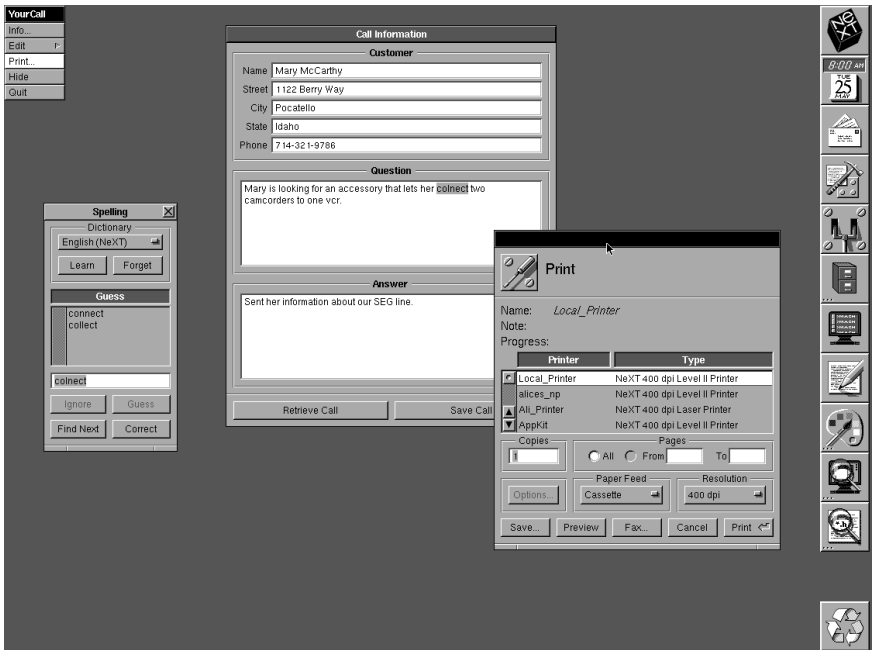


Control-drag from the Spelling... item to the First Responder icon.

With the connection made, double-click the showGuessPanel: action in the Inspector. Then save the YourCall.nib file by typing Command-s.

Testing the User Interface

With the user interface laid out, it's time to test it. Because of NeXTSTEP's object-oriented run-time system, Interface Builder has been able to create the objects making up the user interface as you laid it out. In Test mode, Interface Builder clears its own windows and panels from the screen and presents the objects you've just created as a working interface. To enter Test mode, you select Test from the Interface Builder Document menu. As you test, the objects in your interface send and receive messages, demonstrating both their standard behavior and features you've enabled by simple Interface Builder connections.



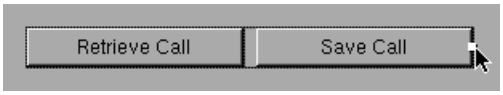
As the illustration suggests, Test mode lets you type in the form, tab between fields, cut and copy text, check spelling, and print or fax the form. Test mode

offers a completely accurate experience of the behavior of objects in the newly designed user interface, independent of the custom code for the application.

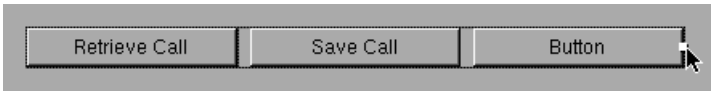
Although you can test the standard NeXTSTEP features while running YourCall in Test mode, the custom features of the application aren't yet available. Saving and retrieving records will be implemented by the yet-to-be-created CallController and CallRecord classes.

Modifying the User Interface

Let's say that as they try the interface for their new application, the Customer Support representatives requests a new feature—a Button to clear the form without saving. To accommodate such a request:



Drag to resize the Buttons



Alternate-drag to add a third Button, then rename it "Clear Form"

In addition to this user interface change, implementing this third Button-controlled feature will require a third CallController method, clearForm:, to actually remove information from the fields in the form.

Interface Builder versus Screen Painters

It's important to note that, unlike development tools for other environments, Interface Builder does not generate code for the user interface. Instead, it creates and saves the actual objects used by your application, using a facility known as *archiving*.

Archiving is a general-purpose Objective C storage feature that allows objects to be written to and retrieved from files. Interface Builder uses archiving to save and retrieve the objects in a user interface. In addition to the objects, the archive includes attributes and connections for those objects. Another use of archiving is the simple data storage and retrieval mechanism implemented by YourCall, described in step 6.

When a NeXTSTEP application starts, it unarchives the objects saved by Interface Builder. This makes Interface Builder more flexible than any screen painter. To adjust the design of the user interface for a NeXTSTEP application at any time, you open the archive in Interface Builder, alter the interface graphically, and save. User interface design is thus independent of code development.

Another distinguishing feature is that Interface Builder lets you add custom objects to the user interface through its CustomView. That is, you can create

your own user interface objects and install them in your application's windows, using a CustomView object as a stand-in. You can also add your own objects to a user interface using *loadable palettes*, an Interface Builder feature described in "Extending the Advantage."

Finally, Interface Builder provides control over more than the user interface, by letting you specify and connect objects with no visible presence in an application—such as the CallController object. This power is demonstrated in the steps that follow.

Creating the User Interface: Summary

In this step, you've seen how Interface Builder lets you create a highly functional user interface by manipulating standard NeXTSTEP objects with a mouse-based graphical editor. Through simple connections, these objects can be "wired up" to provide a menu-based application with many standard features enabled, including text editing, spell checking, printing, and faxing. In Test mode, you saw how Interface Builder can actually activate the objects in the user interface to let them work exactly as they will in your custom application. In the next steps, you'll see how Interface Builder extends its capabilities to the custom objects in your applications.

STEP 4: SPECIFYING CUSTOM OBJECTS

Once the user interface is in place, it's time to develop the two new types of objects required to give YourCall its unique behavior: CallController and CallRecord. In this step, you'll see how to use Interface Builder to begin creating objects. But first, let's take a look at some of the details of how objects are implemented.

Objects and Classes

It should be clear by now that applications are composed of many different kinds of objects and often have more than one object of the same kind. For example, the user interface of YourCall is made up of a Window, a Form, two TextFields, and several Buttons. All these objects are provided by the Application Kit. In terms of custom objects, YourCall will have only one CallController object, but will create a new CallRecord object each time the user saves call information.

Objects of the same type belong to the same class. To define a class, you write code that defines the data and procedures for its objects. In this step, we begin the actual process of defining custom classes for YourCall.

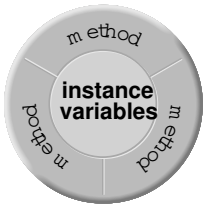
Instance Variables and Methods

Instance variables are the data items belonging to an object. A class definition includes a list of instance variables for its objects. Each object has its own set

of these instance variables. Only an object itself can directly access its instance variables.

Methods are the procedures implemented by an object. A class definition includes the code for the methods used by its objects. The code of a method can implement any functionality, including setting and retrieving the object's instance variables, sending messages to other objects, and so on. A method is invoked by sending a message to an object.

Thus, an object can be thought of as a set of instance variables encapsulated by methods.



Outlets, Targets, and Actions

For one object to send a message to another, it needs to have a way of identifying that object. It also needs to know what methods that object can respond to. *Outlets*, *targets*, and *actions* provide standard ways to provide this information to objects.

In designing the user interface, you saw how to draw simple graphic connections between objects to implement tabbing between fields. You stretched a line from one TextField to the next, then clicked an *outlet* named nextText.

In fact, an outlet is an instance variable that one object uses to identify another. When you connect one TextField as the nextText outlet of another in Interface Builder, you're actually setting an instance variable named nextText. When the user presses the Tab key in one field, that field sends a standard message to its nextText object, telling that object to begin editing. Similarly, when an object you design needs to communicate with another object, you provide the connection by declaring an outlet instance variable.

One special kind of outlet is called a *target*. NeXTSTEP's control objects—Buttons, Sliders, TextFields, MenuCells, and so on—have targets to identify the objects that they send messages to as the user manipulates them. For example, the Print MenuCell in the YourCall menu was connected to the Window object as its target. When the user chooses Print, that MenuCell sends a message to the Window object, telling it to print itself.

A control interacts with its target by sending it an *action* message; the action message tells the target to perform a specific action. In the printing example, the action is smartPrintPSCode:. The Application Kit's Window class implements this method, since it's common to want a Window to print itself in response to user action.

Together, targets and actions provide a standardized way to connect custom objects to the user interface. To create an object to respond to a control in the user interface, you define action methods for it. To connect that object to the user interface, you make it the target of the appropriate control.

Subclasses and Superclasses

Object-oriented programming provides a technique—*subclassing*—that makes it simple to create new classes. To create a subclass, you start with a class whose behavior is closest to that which you want to implement, then add new behavior in the form of new methods and instance variables. The new class is the *subclass* of the original; the original class is the *superclass* of the new. A new class *inherits* all the methods and instance variables defined for the superclass. Because of inheritance, the new class by default has exactly the same behavior as its superclass. A class specification identifies the superclass and adds methods and instance variables to implement its unique behavior.

In addition to implementing new methods, a class can *override* methods it inherits from its superclass. To do so, it includes code for the overridden method in its class definition. When overriding an inherited method, the new class can either implement entirely new behavior, or it can keep the original behavior and expand on it.

Specifying CallController and CallRecord

While the CallController and CallRecord classes were discussed in the design step, at this point it's time to see how the implementation details just described apply to these classes.

First, consider how CallController will use outlets. CallController needs to interact with objects in the user interface, so that it can access data entered by the user and display data retrieved from the data manager. To do so, it needs three outlets—customerForm, questionText, and answerText—corresponding to the text entry components of the user interface. CallController also needs to interact with the data manager object (the HashTable) in order to request storage and retrieval of CallRecords. For this, CallController will use an outlet named callTable.

Next, consider the actions that CallController needs to implement. The CallController object will be the target of the three controls in the YourCall user interface. So, it needs three action methods to respond to these controls: saveCall: and retrieveCall: from the original specification, and clearForm: for the requested enhancement.

CallRecord's task is to store the information for a customer call, so the CallRecord class needs to define instance variables for each data item. It also needs methods for setting and retrieving those data items. As a storage object, CallRecord acts entirely at the behest of other objects in YourCall, so it has no outlets or actions.

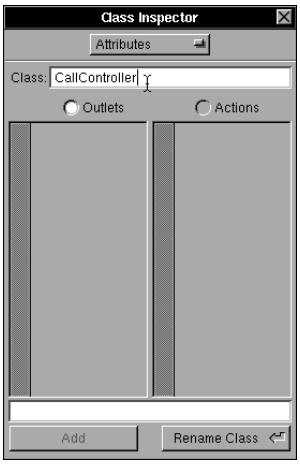
Finally, consider the superclasses for CallController and CallRecord. Both implement very rudimentary, application-specific behavior. As a result, there's not much they could inherit—they'll implement this behavior themselves. The Object class is the most rudimentary class in Objective C; it's the root class for all other classes. As a result, Object is the appropriate class to use as the superclass for both CallController and CallRecord.

Defining Classes in Interface Builder

Interface Builder provides a simple way to begin defining a class. You can declare the superclass for your class, specify outlets and actions for your class, even add objects of your class to the application and connect them to other objects—all from within Interface Builder. Finally, Interface Builder helps you start coding custom classes by generating template files for you to work from.

To begin defining the CallController class, you click the Classes suitcase at the top of the File window shown below, and choose Object in the browser (Object is the superclass for CallController). Then:

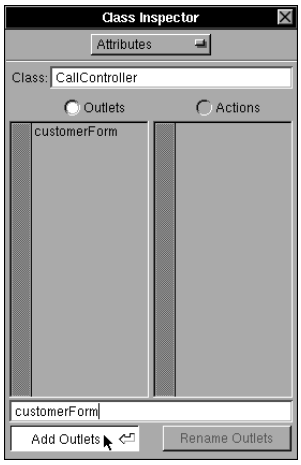
Choose Subclass in the popup list



Type "CallController" in the Inspector and press Return

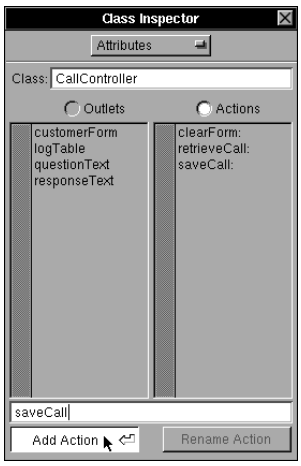
When you enter the new name, the name of the new class in the File window changes to CallController.

To add an outlet instance variable to CallController, use the Class inspector:



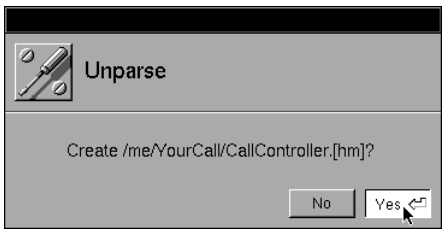
Click Outlets, type “customerForm”, then click Add Outlets

Use the same technique to add questionText, answerText, and callTable outlets. To add action methods to CallController, click the Actions button, and enter saveCall:, retrieveCall:, and clearForm: actions. When you’re finished, the inspector shows all outlets and actions for CallController:

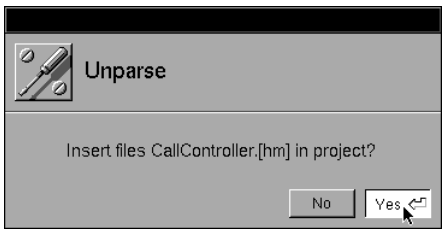


Once the CallController class is specified, you can create its template source files and add them to the project through Interface Builder. The process for creating source files from an Interface Builder specification is called *unparsing*. When the files are unparsed and added to the project, they will be compiled (if they’ve been updated since the last compile) any time you build the project.

Choose Unparse

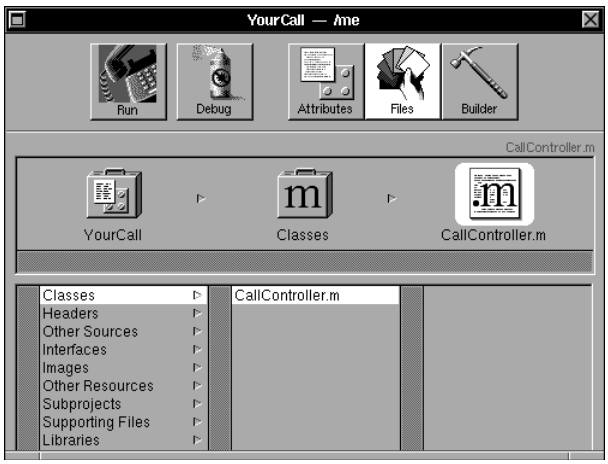


Click Yes to create the files



Click Yes to add the files to the project

From Project Builder's File display, you can see that the class is now included in the project.



By the same means, you can create a template class specification for CallRecord, a subclass of Object. This class has neither action methods nor outlet instance variables, so its behavior will be defined entirely in code that you write. To recap, the steps are:

- Choose Classes in the Files window.
- Choose Object in the class browser.
- Choose Subclass from the popup list.
- Type "CallRecord" in the Class inspector.
- Choose Unparse in the Files window.

Note that since CallRecord isn't part of the user interface, you might create the CallRecord class specification directly, by creating your own text files. However, when you use Interface Builder, it not only creates the template source files, it places them in the appropriate directory and adds them to the project—ensuring that your class is compiled with the rest of your program.

Also note that if you do create a class specification directly in text files, Interface Builder can parse these sources to find and display any outlets and action methods specified. This lets you create a class on your own, then use Interface Builder to graphically add it to your application.

Specifying Custom Objects: Summary

In this step, you've seen how Interface Builder provides ways to add custom objects to the application model. By declaring the outlets and actions for a class of objects in Interface Builder, you create "hooks" to connect those custom objects to the other parts of the application. The next step demonstrates how to create the actual connections that take advantage of those hooks.

STEP 5: CONNECTING CUSTOM OBJECTS

In step 2, you saw how user interface objects are connected using Interface Builder. In effect, you wire up the user interface of an application without writing any code. But Interface Builder recognizes that not all objects in an application are part of the user interface—and provides ways to connect those objects as well.

As one such object, CallController provides data transfer between the user interface and data storage. To do this work, CallController needs to be

connected as the target of three Buttons in the user interface. Its retrieveCall:, clearForm:, and saveCall: action methods are to be invoked when the Buttons are clicked. Now that the CallController is specified, you can use Interface Builder to add a CallController object to the application and connect it to the Buttons.

As a reminder, we still haven't written a single line of code for CallController. That comes in the next step. All we've done is declare outlet instance variables and action methods for CallController and had Interface Builder turn those specifications into template source files. It's also important to note that Interface Builder doesn't generate code to add the CallController object or make its connections in your application. Instead, it creates a stand-in for CallController, and archives the stand-in with other objects in the interface archive file. Later, when YourCall is compiled and run, this stand-in is automatically replaced by an actual CallController object.

Adding CallController to the Application

Though objects like CallController don't appear in an application's user interface, Interface Builder can represent such objects graphically. An object belonging to a specific class is referred to as an *instance* of that class. To add a CallController instance to YourCall in Interface Builder:

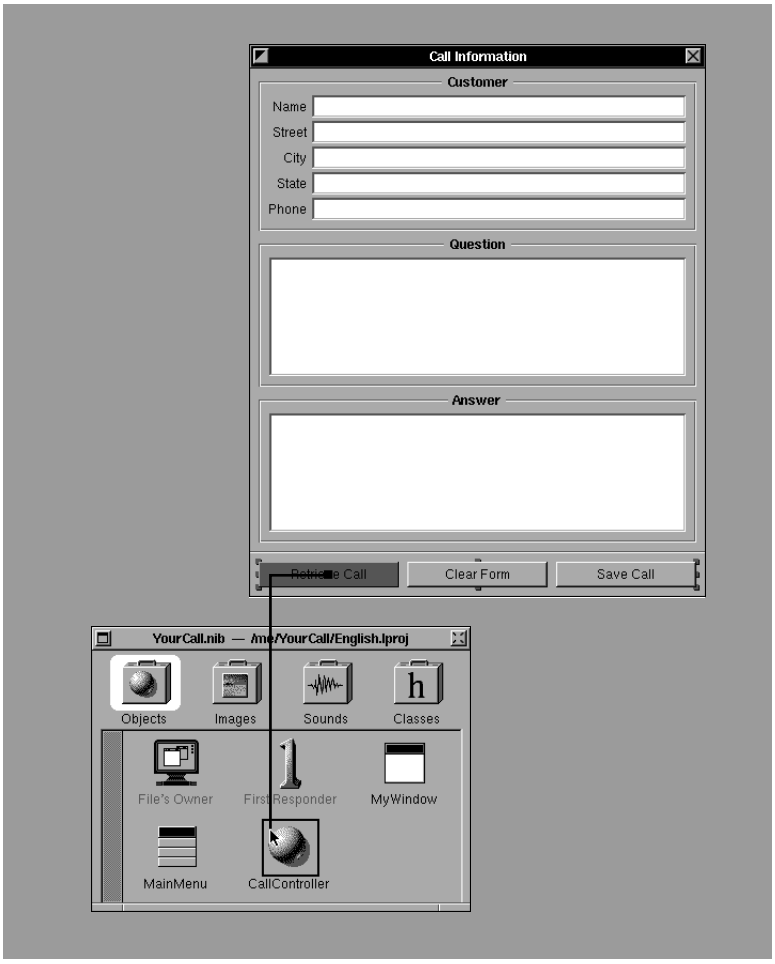
Highlight CallController in the File window, then choose Instantiate

A CallController icon is added

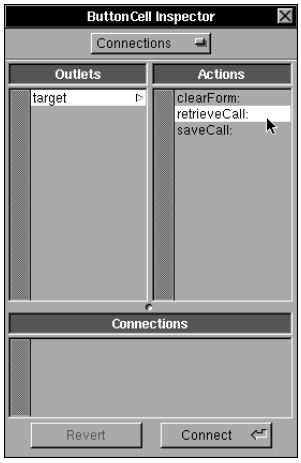
Now you can use the icon to connect this instance of CallController to other objects in the application.

Target-Action Connections

With a CallController icon in the interface, you can connect the Button objects to their target and establish their action methods. For example, to make connections between the Retrieve Call Button and the CallController:



Click the Button and Control-drag to CallController

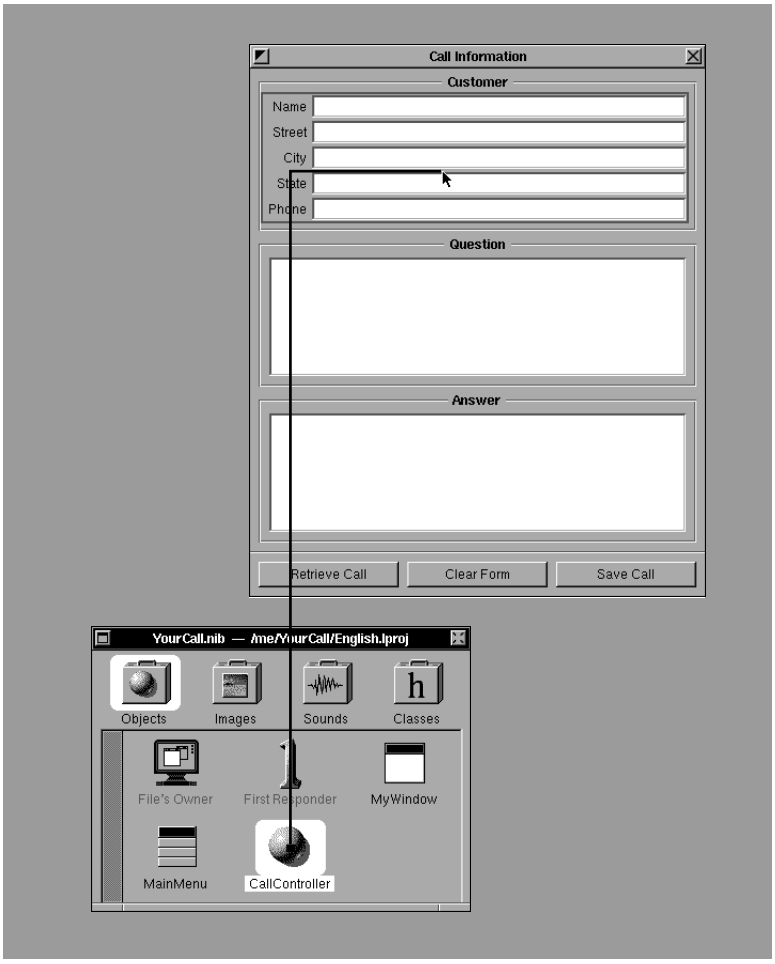


Double-click **retrieveCall:**

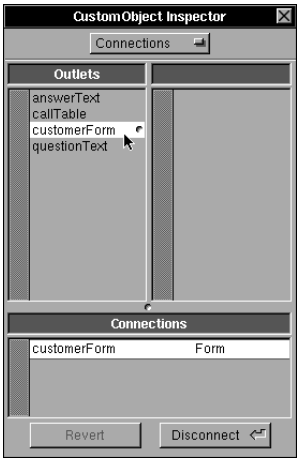
Similarly connect the Clear Form Button to CallController's clearForm: method and the Save Call Button to CallController's saveCall: method.

Outlet Connections

Finally, to access data in the form, CallController needs connections to the Customer Form, the Question TextField, and the Answer TextField. These are made by setting CallController's customerForm, questionText, and answerText outlet instance variables. Starting with the outlet for the Customer form:



Click CallController and Control-drag to the Form
(Be sure to connect to the whole Form as shown,
not just one field)



Double-click **customerForm**

Similarly, set the Question TextField as CallController's questionText outlet and the Answer TextField as CallController's answerText outlet. The callTable outlet will be set later in CallController's code.

This done, the connections in YourCall are complete. Save the YourCall.nib file again by pressing Command-s.

Connecting Custom Objects: Summary

In this step, you've seen how Interface Builder, working with a simple class specification, can add instances of that class to your application and connect them to other objects. This demonstrates the flexibility of NeXTSTEP's object-oriented application framework, which provides a number of standardized connections for custom objects. It also demonstrates how Interface Builder provides simple, graphical ways to make those connections. In the next step, you'll see how to implement the code that takes advantage of these connections to give an application its unique behavior.

STEP 6: IMPLEMENTING CUSTOM BEHAVIOR

A class definition consists of the code declaring and implementing object behavior by means of instance variables and methods. A class is defined in two files, the interface file and the implementation file. When you specify a class in Interface Builder, it will create templates for both the interface and implementation files.

The *interface file* is identified with a ".h" (for *header*) extension. The interface file declares instance variables and methods for a class. This file is referred to as the interface file in standard C terminology because it presents the public declaration of the class. A class interface file is referenced (or *imported*) by other code files that send messages to objects of that class.

The *implementation file* is identified with a ".m" (for *methods*) extension. The implementation file contains the program code for the class. The code can include a combination of Objective C, C, and C++ syntax.

In this step, you'll see how to write the actual code for the custom classes of YourCall. You won't find all the code here (that's in the back of the book, for your reference). What you'll see should give you a taste of Objective C coding style and demonstrate how NeXTSTEP continues to support your application development efforts as you're coding.

The Syntax of a Message

An Objective C message expression has two components: an object and the message. Objective C messages use the following syntax:

```
[receiver methodName:value]
```

In this example, *receiver* is the object receiving the message. *methodName:* is the name of the method invoked by this message. *value* is an argument to the

method. In this case, the method has just one argument, thus the method name includes a single colon. Some methods may have no argument (and thus no colons). Other methods may have several arguments, each preceded by a colon and frequently a keyword before each colon. You'll see examples of these conventions in the code that follows.

Implementing CallRecord

Remember that each CallRecord object is intended to represent an individual call. The CallRecord class will specify one instance variable for each field of data for a call; however, that's an internal implementation detail. More importantly, CallRecord will specify and implement methods to save and retrieve that data, such as setName: and name.

CallRecord also needs a way to save and retrieve its objects in a file. Objective C provides this through its archiving mechanism—the same mechanism used by Interface Builder to store user-interface objects. Two methods, read: and write:, are the standard methods for archiving objects. Declared by the Object class, these methods are overridden by each subclass to enable its objects to be saved and retrieved.

The read: and write: methods illustrate the power of *polymorphism*, the object-oriented feature in which different classes implement the same methods so that each can respond to the same message in its own way. Polymorphism ensures consistent behavior among objects of diverse classes. As you've seen, Interface Builder can save Windows, Menus, Forms, TextFields, and other objects in an archive file because each of their classes implements its own version of read: and write:.

Declaring CallRecord

CallRecord.h is the interface file for CallRecord. This file declares instance variables used and methods coded in CallRecord.m, the implementation file. Here's the template interface file provided by Interface Builder:

```
#import <appkit/appkit.h>

@interface CallRecord:Object
{
}


@end
```

First, note that a class specification refers to other classes by referring to their header files in a #import line. The template file provided by Interface Builder imports the header file appkit/appkit.h, which refers to all classes in the Application Kit and other core components of NeXTSTEP. Classes created

with Interface Builder are assumed to need to interact with these NeXTSTEP components.

Because no instance variables or methods were declared in Interface Builder, none appear in CallRecord's template header file. To declare CallRecord, we'll add instance variable and method declarations to this file.

The data collected for a call is a set of text fields. Each field can be represented by an instance variable of the standard C-language type for character strings: char *. CallRecord has seven of these instance variables: name, street, city, state, phone, question, and answer.

The methods for setting and getting the data should be consistent from field to field. For example, the method for setting the customer name field would be declared as:

```
- setName:(const char *)theName;
```

setName: is the method name. The type of the argument is declared as const char *. The argument name is theName; this name will be used within the code of the method to refer to the argument.

The argument type, const char *, is consistent with CallRecord's instance variables and with the data that can be retrieved from objects in the user interface. The C-language qualifier const assures that the method can't change the character string passed to it.

The method for retrieving this field is declared as:

```
- (const char *)name;
```

Here, const char * declares the return type of the method, and name is the name of the method.

The return type also includes a const qualifier to assure that an object sending a name message to a CallRecord is given read-only access to the data. The data is thus protected from outside tampering.

Here is the completed header file for CallRecord containing all instance variables and method declarations:

```
#import <appkit/appkit.h>
@interface CallRecord:Object
{
    char *name;
```

```
    char *street;
    char *city;
    char *state;
    char *phone;
    char *question;
    char *answer;
}

- (const char *)name;
- setName:(const char *)theName;
- (const char *)street;
- setStreet:(const char *)theStreet;
- (const char *)city;
- setCity:(const char *)theCity;
- (const char *)state;
- setState:(const char *)theState;
- (const char *)phone;
- setPhone:(const char *)thePhone;
- (const char *)question;
- setQuestion:(const char *)theQuestion;
- (const char *)answer;
- setAnswer:(const char *)theAnswer;
- read:(NXTypedStream *)theStream;
- write:(NXTypedStream *)theStream;
- free;

@end
```

Note that because read: and write: are declared in Object, CallRecord's superclass, they aren't required in the header file: CallRecord inherits them. They're here by convention to indicate that CallRecord implements its own version of them. Note also the free method. It's also inherited from Object, and overridden by CallRecord to ensure that memory claimed by the instance variables will be properly freed.

Defining Data Access Methods

CallRecord.m is the implementation file for the CallRecord class.

The methods for adding data to a record use fundamentally the same code. Here's the implementation of setName: as an example:

```
- setName:(const char *)theName
{
    if (theName) {
        name = NXCopyStringBuffer(theName);
    }
    return self;
}
```

This code uses an if statement to test whether the argument, *theName*, is valid. If so, it uses the NeXTSTEP function NXCopyStringBuffer() to copy the argument and assign the copy to the instance variable name. Note that within its code, CallRecord can access instance variables such as name directly.

The return value, self, is a variable representing the object receiving the message. Much as outlets are used to identify other objects, self is used to

identify an object within its own code. This is useful when an object needs to send messages to itself. self is the default return value for Objective C methods.

The methods for accessing the data in a CallRecord are even simpler. Here's the name method as an example:

```
- (const char *)name;
{
    return name;
}
```

This very minimal code simply returns the instance variable—name—as a constant string.

Defining Data Archiving Methods

CallRecord's mechanism for storing and retrieving data on disk is implemented with two methods, read: and write:, inherited from Object. These methods are part of the standard Objective C archiving facility.

Archiving takes advantage of a NeXTSTEP feature known as *typed streams* to store and retrieve objects. A typed stream is a data buffer that contains information about the data types stored within it; the buffer can be copied in memory and written to or read from a file using standard typed stream functions. When storing a set of objects, a typed stream is opened, each object is sent a write: message to write its instance variables to the typed stream, then the stream is written to a file and closed. To retrieve, a typed stream is opened on a file, each object in the file is sent a read: message to read its instance variables from the stream, then the stream is closed.

The implementation of CallRecord's write: method is:

```
- write:(NXTypedStream *)stream
{
    [super write:stream];
    NXWriteTypes(stream, "*******", &name, &street, &city,
        &state, &phone, &question, &answer);
    return self;
}
```

The sole argument to the write: method is the typed stream for the object to write itself to. The method implementation begins by sending a write: message to super. super is a variable which acts as an object of the superclass—Object in the case of CallRecord. This message ensures that inherited instance variables are written to the typed stream before variables declared in CallRecord.

Next, the implementation invokes the NXWriteTypes() function—the function that actually writes the instance variables to the stream. There are three types of arguments to this function. stream is the typed stream where the

object is being written. The string "*******" is a format string indicating that the seven instance variables to be archived are character strings. Finally, &name and subsequent arguments represent the addresses of those instance variables.

The implementation of the read: method is directly parallel to the write: method.

Implementing CallController

Now, let's turn to the implementation of CallController. It's implemented through two files—the interface and the implementation file.

The CallController Interface File

Like CallRecord, CallController's interface and implementation file are created through Interface Builder. However, in the case of CallController, the interface file already contains the outlets and actions declared in Interface Builder. There are a few additions to be made to this basic file, indicated in bold in the following text:

```
#import <appkit/appkit.h>
#import "CallRecord.h"

@interface CallController : Object

{
    id customerForm;
    id questionText;
    id answerText;
    id callTable;
    char callFilePath[MAXPATHLEN + 1];
}

- init;
- awakeFromNib;
- clearForm:sender;
- retrieveCall:sender;
- saveCall:sender;
- free;

@end
```

The instance variable callFilePath is a character array that will be used to keep track of the file where the CallRecords are stored. The file referenced by this variable will be opened when the application starts so that records may be read. It will be opened again, by the saveCall: method, to update the contents of the file each time a new record is added to the database.

The init method, inherited from Object, is used to perform any initialization required to make an object fully functional—for example, to establish internal connections with other objects, to set instance variables, and so on. The

awakeFromNib method is sent to all objects as they are unarchived; in CallController, this method prepares objects connected to CallController for user input. The free method, like that for CallRecord, is used to free storage allocated by CallController.

Initializing the CallController Object

Earlier, you saw how to add an object representing CallController to YourCall from Interface Builder. When the application starts, the Application object unarchives the Interface Builder file, allocates an instance of CallController, then sends that CallController an init message.

As go-between from the user interface to the data storage, CallController needs to establish connections to both. Connections with the user interface are made in Interface Builder and archived with the user interface. The connection with the data manager needs to be made by CallController itself, within its init method. CallController.m implements the init method as follows.

```
- init
{
    NXTypedStream *callStream;
    BOOL fileFound;

    [super init];
    fileFound = [[NXBundle mainBundle]
        getPath:callFilePath forResource:"call" ofType:"log"];
    if (!fileFound) {
        strcat(callFilePath,"/call.log");
    }
    callStream =
        NXOpenTypedStreamForFile(callFilePath, NX_READONLY);
    if (callStream) {
        callTable = NXReadObject(callStream);
        NXCloseTypedStream(callStream);
    }
    else {
        callTable =
            [[HashTable alloc] initKeyDesc:"*" valueDesc:"@"];
    }
    return self;
}
```

This method begins by declaring two local variables. (Local variables, a standard C feature, can be referenced only within the function or method where they're declared.) callStream is used to refer to the stream for reading the HashTable and its contents. fileFound is a boolean used to test for the presence of the file in the appropriate place.

The first statement sends an init message to super, which ensures that any initialization specified for the objects of the superclass will be performed.

YourCall uses a NeXTSTEP feature called *bundles* to manage its data file. The NXBundle class provides an object-oriented approach to application resource file management. Resources files are accessed by bundles rather than file

directories. This frees NeXTSTEP applications from dependence on a particular file system.

A NeXTSTEP application consists of a main bundle (the main directory) containing resource files and other bundles. The most important application resource, the executable, is contained in the main bundle. The user interface archive file is placed in a bundle (or subdirectory) within the main bundle.

The database file, call.log, will be stored in and retrieved from the application's main bundle. The init method locates this file and stores its path in an instance variable callFilePath with the following code.

```
[[NXBundle mainBundle]
    getPath:callFilePath forResource:"call" ofType:"log"];
```

This is a *compound message statement*, and it demonstrates a powerful shorthand provided by Objective C. Let's look at the inner message statement first:

```
[NXBundle mainBundle]
```

This simple statement returns an object representing the application's main bundle. To that object, the outer part of the statement sends a message getFile:forResource:ofType:, which requests that the main bundle object find the path for the file named call.log. If the outer message statement finds the file, it returns YES and places the full path to that file in the variable callFilePath. If not, it returns NO and places only the path to the main bundle in the variable. The return value is assigned to the local variable fileFound.

If the file isn't found, the code uses a C function, strcat(), to put the full file path in the variable callFilePath variable. (This path is used by the saveCall: method to save the callTable in the file.)

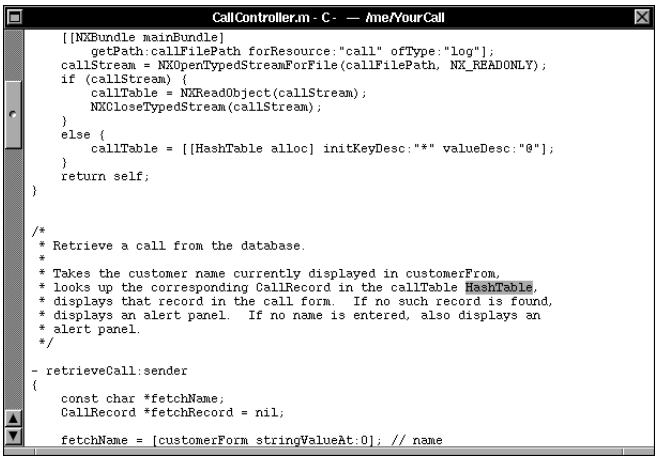
The code then attempts to open a typed stream on the file using the function NXOpenTypedStreamForFile(). If it succeeds, it retrieves the HashTable object from the file and assigns it to the callTable instance variable using the NXReadObject() function. This NeXTSTEP function unarchives the HashTable by invoking its read: method, which in turn invokes read: on each of the CallRecord objects contained in the table.

If a typed stream can't be opened on the file, this method allocates and initializes a new HashTable for use by the CallController.

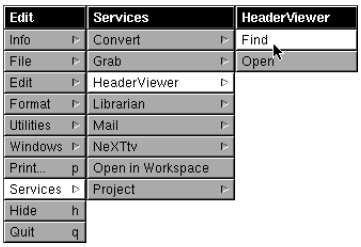
Retrieving Data

The retrieveCall: method looks up a CallRecord in the HashTable when the user enters a name in the Name field and presses the Retrieve Call Button. This method is implemented using HashTable's ability to locate a data value by a particular key item. But how does HashTable implement this ability?

To help answer such questions, NeXTSTEP provides Header Viewer, a tool for class browsing and access to documentation. Say, for example, you want to find the HashTable method needed to implement retrieveCall:. You can easily do so while editing the code file from the Edit application, using the NeXTSTEP Services facility:

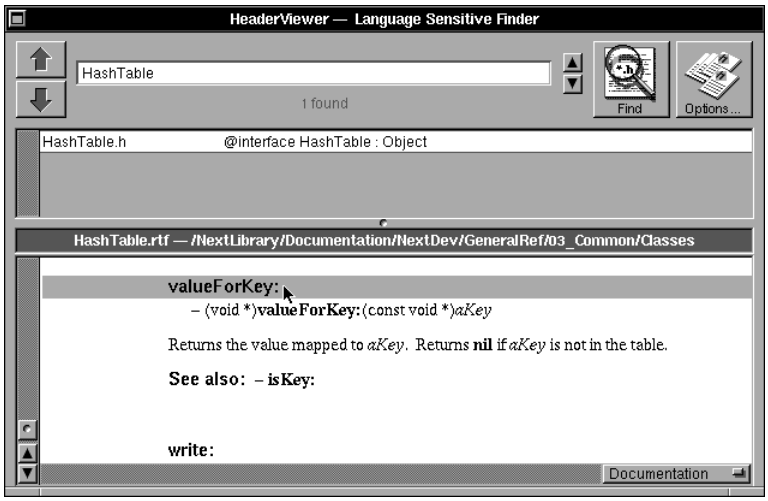


Highlight the class name



Choose HeaderViewer in the Services menu

When the Header Viewer window appears, it displays the on-line documentation for HashTable. You can scan through the documentation to locate the appropriate method. From the same window, you can also view the interface file for the HashTable class, and browse other NeXTSTEP classes.



Using the HashTable method valueForKey:, the following code implements CallController's retrieveCall: method:

```
- retrieveCall:sender
{
const char *fetchName;
CallRecord *fetchRecord = nil;

fetchName = [customerForm stringValueAt:0];
if (fetchName && strlen(fetchName)) {
  fetchRecord = [callTable valueForKey:fetchName]
  if (fetchRecord){
    [customerForm setStringValue:[fetchRecord street] at:1];
    [customerForm setStringValue:[fetchRecord city] at:2];
    [customerForm setStringValue:[fetchRecord state] at:3];
    [customerForm setStringValue:[fetchRecord phone] at:4];
    [questionText setStringValue:[fetchRecord question]];
    [answerText setStringValue:[fetchRecord answer]];
    [customerForm selectText:self];
    }
  else {
    NXRunAlertPanel("Search Failed", "Customer %s not found",
      NULL, NULL, NULL, fetchName);
    }
  }
else {
  NXRunAlertPanel("Search Failed",
    "Please enter a customer name", NULL, NULL, NULL);
  }
return self;
}
```

Note first the declaration for this method:

```
- retrieveCall:sender
```

Recall that this method is an action method, defined from Interface Builder and used as the action of the Retrieve Call Button object. The single argument to this method—and all action methods—is *sender*. *sender* identifies the control

object sending the message. This argument can be used by a receiver such as CallController to query the control for anymore information it may need. For example, if the control were a slider, the method could send a message back to the slider asking for its current position. In this case, no further information is required. The user's intentions are made clear by the action of pressing the Button.

This method declares two local variables. fetchName is a character string for the name in the form. fetchRecord is a CallRecord used to refer to the record retrieved from the database.

The first line of code sends a message to the customerForm requesting the string in its first field—Name—and assigns the return value in the variable fetchName (by C language convention, the index of the first field is 0).

```
fetchName = [customerForm stringValueAt:0];
```

If the field contains an entry, this method searches for the corresponding CallRecord in the callTable, using the valueForKey: method located with Header Viewer:

```
fetchRecord = [callTable valueForKey:fetchName]
```

If the search finds a record, the subsequent code queries that record for its data and places it in the form. The code that performs this question and display again illustrates the use of a message within a message:

```
[customerForm setStringValue:[fetchRecord street] at:1];
```

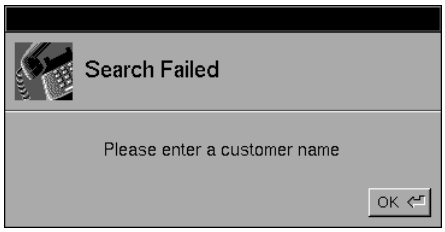
The inner message statement gets the street stored in the CallRecord. This return value is passed as an argument to Form's setStringValue:at: method, placing the street in the second field of the Form. After all data is placed in the form, the first field (Name) is selected with the code:

```
[customerForm selectText:self];
```

The last few lines of code offer alternative interaction with the user. If no name appears in the form, or if no record is found for the name, the method displays attention panels to indicate the situation. In both cases, displaying the attention panel requires just one line of code. For example, if no name is entered, this code is invoked:

```
NXRunAlertPanel("Search Failed",
  "Please enter a customer name", NULL, NULL, NULL);
```

The attention panel provided by this function is a standard part of the user interface, used by many NeXTSTEP applications. The NXRunAlertPanel() function enables you to customize the message and the buttons displayed by the panel.



When the application executes this function, it automatically stops and waits until the user clicks the OK button, then continues. Features such as this help ensure that your applications share a common interface and consistent behavior with other NeXTSTEP applications.

CallController's saveCall: method uses similar code to create a new CallRecord, copy the data from the form to the record, place the record in the HashTable, and write the HashTable to the call.log file.

Implementing Custom Behavior: Summary

In this step, you've had a brief glimpse of the code used to implement an Objective C class. Perhaps more important than the coding details presented here are two general points. First, Objective C syntax presents a straightforward yet flexible style for creating object-oriented code. Second, NeXTSTEP continues to support your development efforts as you code, by providing program functionality such as archiving, bundles, and attention panels, and programming tools such as Header Viewer. In the final step, you'll see how other NeXTSTEP tools help you complete your application development tasks.

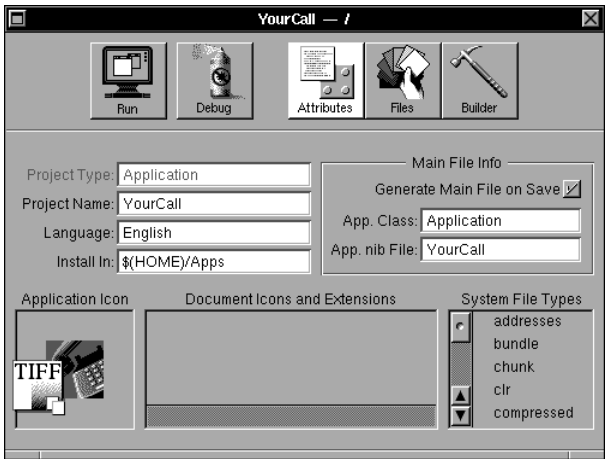
STEP 7: BUILDING AND DEBUGGING THE APPLICATION

Once the custom code is implemented for YourCall, the final step in the development cycle is to make and debug the application. Project Builder's project management features and NeXTSTEP's collection of application development tools are designed to help you in the process of putting the finishing touches on your applications.

Adding an Application Icon

As you've seen, an application's icon is a standard part of its user interface. Icons represent applications in the Workspace Manager, providing the user with a consistent way to start NeXTSTEP applications.

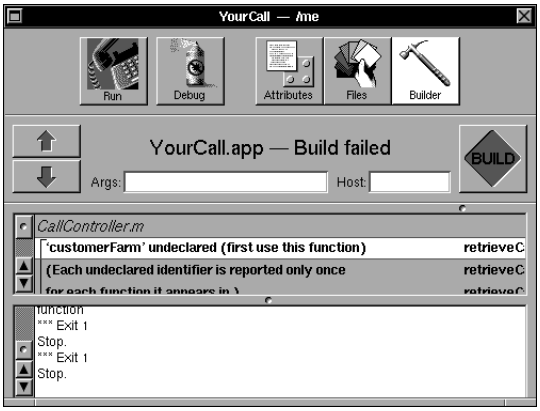
The NeXTSTEP Icon Builder application lets you create the icon for an application. Once it's created, you associate an icon with the application by dragging the file into the Application Icon well in the Project Builder Attributes display.



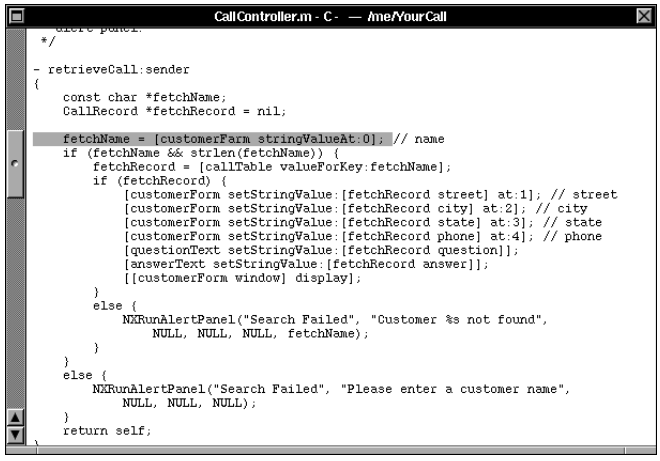
Building the Application

As part of its project management facilities, Project Builder maintains the Makefile containing information on how to build your application from its sources. When Project Builder builds your application, it uses this file and the standard UNIX make program to build the components of your application. The make program compiles the project's code files and links them into the executable file. It also copies other resource files—such as the user interface archive file—to the appropriate places in the application.

As your application builds, Project Builder shows compiler and linker progress messages in its Build display, and provides an interactive way to track and edit any coding errors:



Click an error message in the Build display



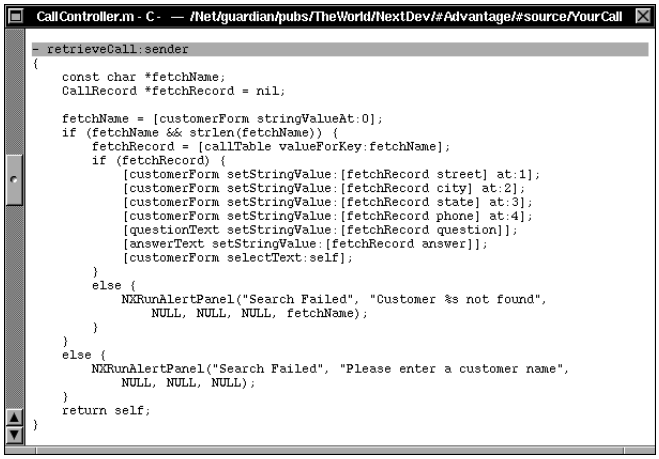
Edit starts and highlights the line containing the error

With the code fixed, you can recompile successfully

Once your application compiles successfully, Project Builder helps you run and debug it.

Debugging the Application

NeXTSTEP's text editor and debugger are integrated with Project Builder to facilitate setting break points, stepping through the application, and tracking down bugs. Using Project Builder's Debug feature, you can run the program interactively, trace errors, and make code fixes. To set a break point, you open the source code file, then:



Click on a break point

Click Gdb in Edit

Click Break At

With breakpoints set, you can then run the application, stepping through the source code in Edit. The GDB control panel lets you set multiple breakpoints, examine the program stack, check variable assignments, and perform other debugging tasks.

Other Development Tools

In addition to Project Builder and Interface Builder, NeXTSTEP provides a variety of applications to help you create, debug, and tune your program.

AppInspector

AppInspector lets you browse through a running application's objects, examine the variables belonging to each one, and look at data represented in those variables. Since outlets (connections to other objects) are among an object's variables, AppInspector enables you to examine the entire network of interconnected objects belonging to an application. Using AppInspector, you can be sure that the objects are connected through the correct outlets, and that their other variables are set to the correct values.

Performance Tuning Tools

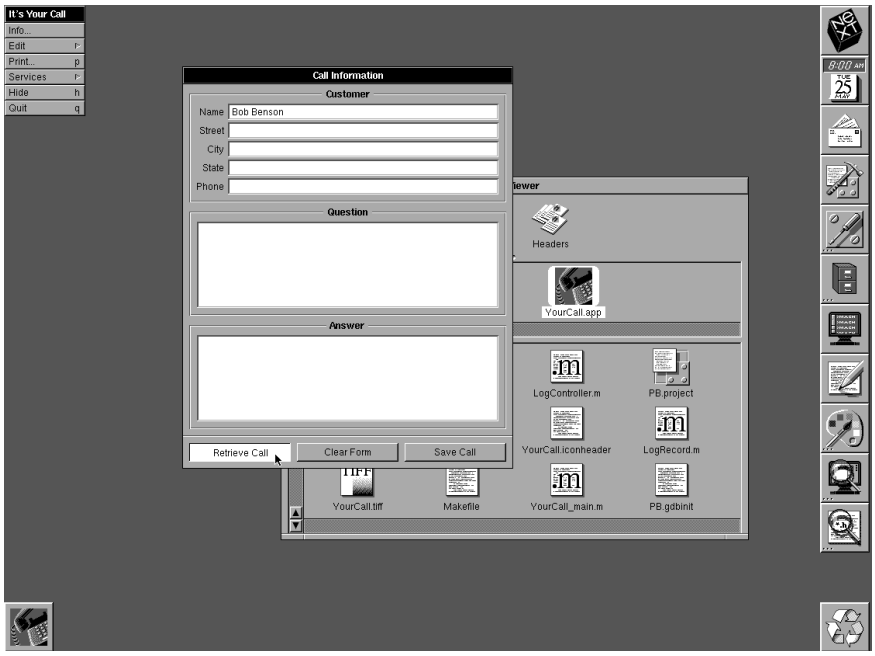
Graphical programs on multitasking platforms place high demand on the facilities of the CPU and operating system. Given this demand, NeXTSTEP provides of tools and techniques for enhancing the performance of an application.

MallocDebug helps achieve optimum memory use by letting you look at exactly how an application uses memory. ProcessMonitor provides ways of inspecting a number of internal details of program operation. For better performance from the virtual memory manager, a link editor feature known as link optimization lets you place code in the executable file so that the most frequently accessed procedures are located near one another.

"NeXT Development Tools" later in this guide provides more on tuning a NeXTSTEP application.

Running the Application

When all the details are in place and the application is built a final time, YourCall is ready to use. Like all other NeXTSTEP applications, it's represented in the Workspace Manager by its icon. All you have to do is double-click to start it:



Building and Debugging: Summary

NeXTSTEP provides an interactive environment for building and debugging your custom applications. Project Builder manages source files and gives fast access to those files when code fails to compile correctly. The debugger and text editor cooperate to provide a quicker find-fix-compile cycle. Other NeXTSTEP development tools give insights into the internal workings and performance of your applications. Thanks to these tools, NeXTSTEP helps you create robust, high performance applications.

SUMMARY

This discussion has presented a guided tour of the process of putting together a NeXTSTEP application. As you've seen, NeXTSTEP simplifies application development in a number of ways.

- Interface Builder provides a graphical approach to assembling the components of an application, while Project Builder provides a control center for source file management and project development.
- The Application Kit offers building blocks that simplify the process of creating your custom applications.
- Object-oriented programming helps you define custom behavior with a minimum of unique code.

The next section of this guide takes the simple YourCall application and enhances it in several ways, demonstrating both the extensibility of the object-oriented programming model and the advanced functionality provided by NeXTSTEP.