Now that you've seen how to develop a simple application in NeXTSTEP, it's time to look at some of the more powerful features available to developers—the Database Kit, custom palettes for Interface Builder, inter-application services, and inter-application communication with Distributed Objects.

As implemented in the previous section, YourCall offers a good example of basic NeXTSTEP programming. However, its behavior is very rudimentary—which makes it the ideal candidate for some enhancement.

Because of NeXTSTEP's object-oriented foundations, it's easy to add functionality to an application. NeXTSTEP's kits add objects for database access, inter-application communications, 3D graphics, and more, in a way that's consistent with the fundamental features. Third-party developers also offer objects and kits that provide useful enhancements. This modular approach helps programmers who know the basic paradigms and tools of NeXTSTEP expand their skills to new domains.

In this section, you'll see how to use several of the advanced features of NeXTSTEP:

- You'll see how the Database Kit simplifies the development of production applications built on commercial database management systems.
- You'll see how to add custom palettes to Interface Builder, letting you reuse your custom objects graphically.
- You'll see how NeXTSTEP applications provide services to one another.
- You'll also see how Distributed Objects provide an object-oriented way to implement peer-to-peer and client-server communication between applications.

Each of the examples that follows starts with the simple YourCall application and uses object-oriented techniques to improve its design and expand its capabilities. The implementations are described only briefly; for more details, see the code listings at the end of this guide.

DATABASE ACCESSS WITH THE DATABASE KIT

In contrast with the simple data management scheme in the YourCall application, most corporate data is maintained on large client-server database systems such as those from Oracle, Sybase, and other vendors. These systems offer the flexibility and power to manage complex sets of interrelated data: customer lists, inventories, payables, receivables, and so on.

Many of these systems offer *forms tools* and *fourth-generation languages* (or *4GLs*) to simplify database application development. But the restrictions presented by such tools frequently diminish their advantages. Forms-based applications don't integrate easily with other applications. Most 4GLs don't have the flexibility or power of full-featured programming languages. And all such tools are database-specific. If you change the underlying database management system, you have to rework all your applications.

The NeXTSTEP Database Kit addresses these and other problems of database application development.

The Database Kit is fully integrated with NeXTSTEP. Like all NeXTSTEP applications, your database applications benefit from a consistent user interface. They share files through the Workspace Manager, services through the Services menu, and data through the Pasteboard. Under NeXTSTEP, custom applications have equal footing with commercial applications.

The Database Kit is database independent. You can move applications from one database management system to another simply by replacing the adaptor that acts as the interface between the Database Kit and the database server. Adaptors for Oracle and Sybase are provided with NeXTSTEP, while adaptors for other databases are available from other vendors.

The Database Kit utilizes a fully-functional programming language. By integrating Objective C, C++, and ANSI-C, NeXTSTEP gives you a powerful combination of object-oriented and standard C language programming.

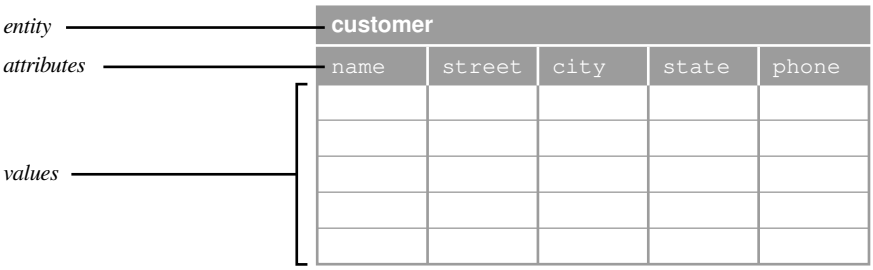
Finally, the Database Kit is integrated with the rest of the NeXTSTEP development environment. DBModeler is a NeXTSTEP application that provides a graphical way to model the organization of data in a database. Through a special Database Kit palette, Interface Builder knows how to incorporate database models and lets you connect to the underlying database server. You can even query a database and fill your user interface with actual data while in Interface Builder's Test mode.

Because of these features, the Database Kit provides an efficient, consistent way to develop mission-critical custom applications that access industrial-strength database management systems.

The Entity-Relationship Model

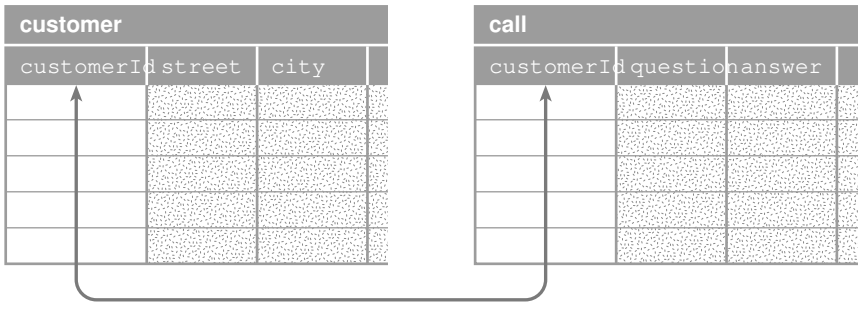
The Database Kit is designed to look at the organization of data as an *entity-relationship model*. In such a model, data is organized into fundamental *entities*, each entity is defined by its component *attributes*, and connections, or *relationships*, are used to link entities. This model applies equally well to relational, flat file, hierarchical, and object-oriented databases.

For example, in a relational database, an entity corresponds to a table; its attributes correspond to the table's columns. A customer table would be one such entity, while its columns name, street, city, and so on, would be its attributes.



The simplest data model is a single entity. You can create more complex models by creating relationships between entities.

Taking the simple example of YourCall, the application really stores two kinds of data: customer data (name, street, city, state, etc.), and call data (question, answer). The design of CallRecord clumps these together. But what if one customer calls several times, while another never calls? To improve storage efficiency and reduce errors, you'd want to store customers and calls in two separate entities. The relationship between these two entities would assign a specific customer to each call. To establish this relationship, you'd include a common attribute such as customer number in each entity.

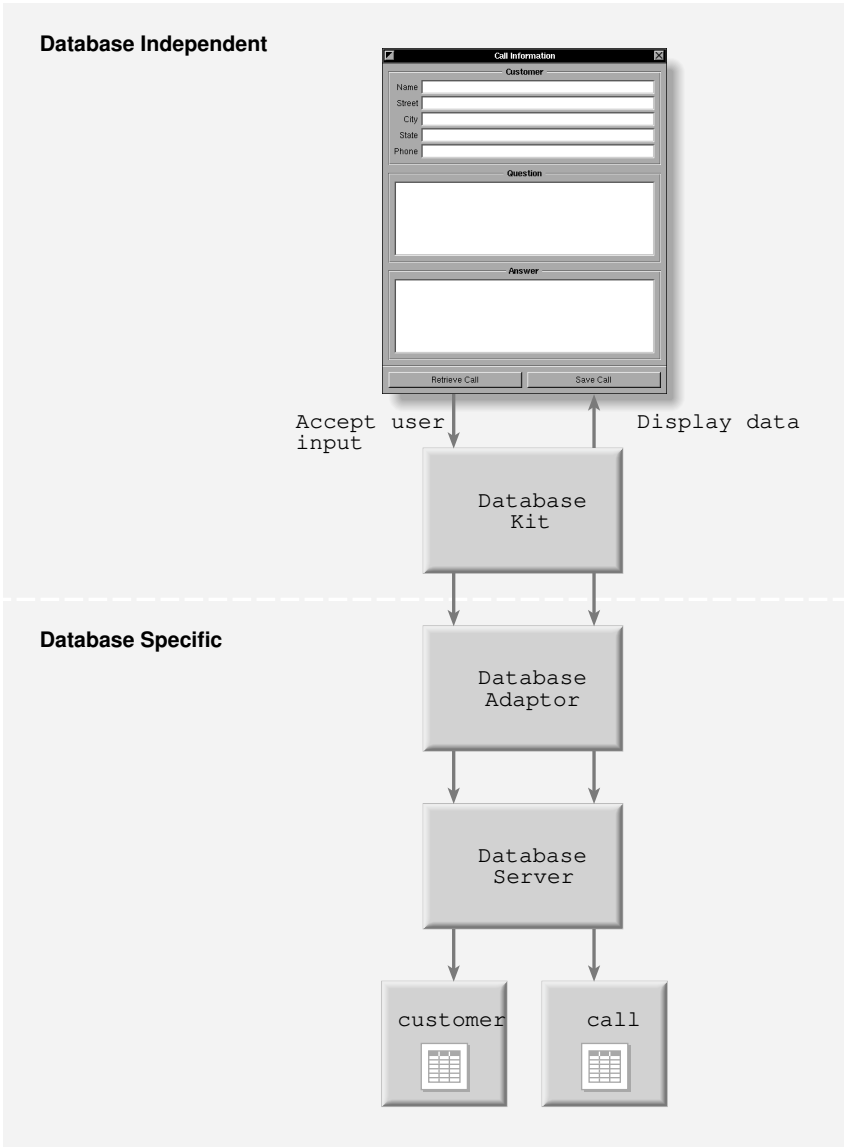


Designing YourCall With the Database Kit

YourCall is simple by design. It uses a flat data model and provides only limited user access to records. Most users would ask that an application like

YourCall give them more, such as the ability to browse through customer records.

Using the Database Kit, this type of feature can be added without a single line of source code and tested against the actual database in Interface Builder's Test mode. Thus the redesign of YourCall is based entirely on the Database Kit, slightly modifying the original interface but depending on the Kit for all other functionality.



Designing The Database

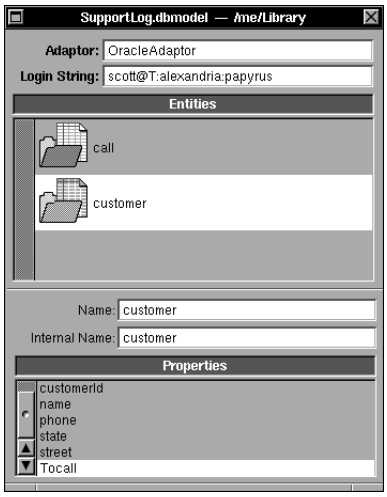
At the bottom level, redesign of YourCall requires a database containing the two entities described above: customer and call. In most companies, the customer entity would probably exist already, with the attributes customerId,

name, street, city, state, and phone (and possibly others). The database entity call may need to be created with the attributes callId, customerId, topic, question, and answer. The topic attribute is added to provide a brief summary of the call, which is useful to identify calls in a browser. The customerId attribute provides the mechanism for relating the call entity to the customer entity.

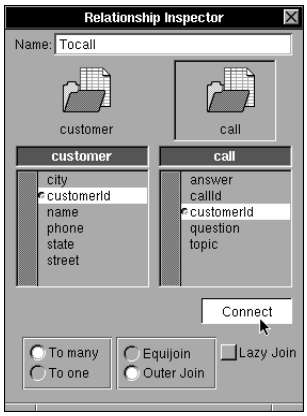
Creating the Data Model

Like CallController in the earlier version of YourCall, the Database Kit provides a data transfer layer between the database and the user interface in an application. Unlike the CallController, the Database Kit provides general purpose transfer layer objects, capable of interpreting a *database model* and translating that model into specific Database Kit objects for representing entities, attributes, and data in the database. So before you create a Database Kit application, you create a database model with DBModeler.

DBModeler is a NeXTSTEP application for specifying relationships between entities in a database. DBModeler accesses information in the database to determine entities and their attributes. It provides this information to the developer in a graphical way.



DBModeler provides a simple interactive process for modeling relationships among the entities in a database. To create a relationship, you simply identify entities and attributes in its Relationship Inspector, then click to connect:



In this case, the relationship is on the attribute customerId in both the customer and call entities. This relationship is used to lookup the calls associated with a particular customer.

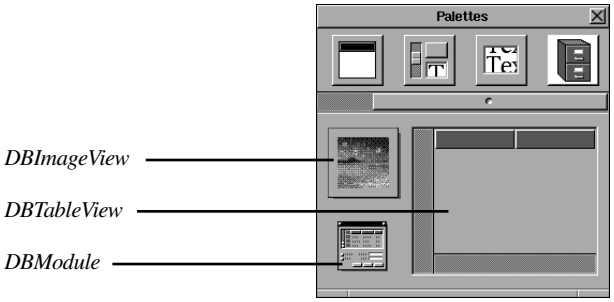
Once a model like this is created, you can use it to help implement the user interface.

Creating and Testing the User Interface

While the Database Kit is designed to work closely with Interface Builder, the two are distinct components of NeXTSTEP. However, using Interface Builder's *custom palettes* feature, you can add a set of Database Kit objects to Interface Builder—while it's running and without any modification to its code. You'll see how to create custom palettes in the next section.

To load a custom palette, you simply choose Load Palettes from Interface Builder's Tools menu. The DatabaseKit.palette file is located in the directory /NextDeveloper/Palettes.

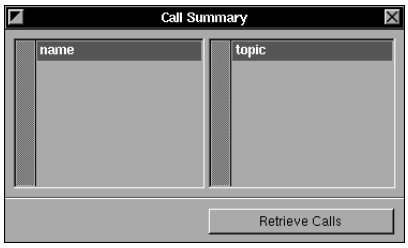
When the palette is loaded, its button appears in the Palettes display. When you click its button, you see that the palette adds three objects to Interface Builder:



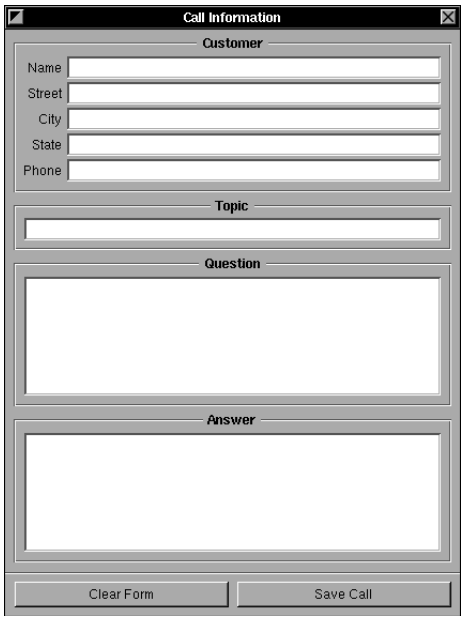
DBModule reads model files into NeXTSTEP applications. DBTableView implements data browsing. DBImageView presents visual data.

In its Database Kit implementation, the interface for YourCall is modified in a couple of ways. First, there's a second window, Call Summary, containing a pair of DBTableViews to implement customer/call browsing. The master

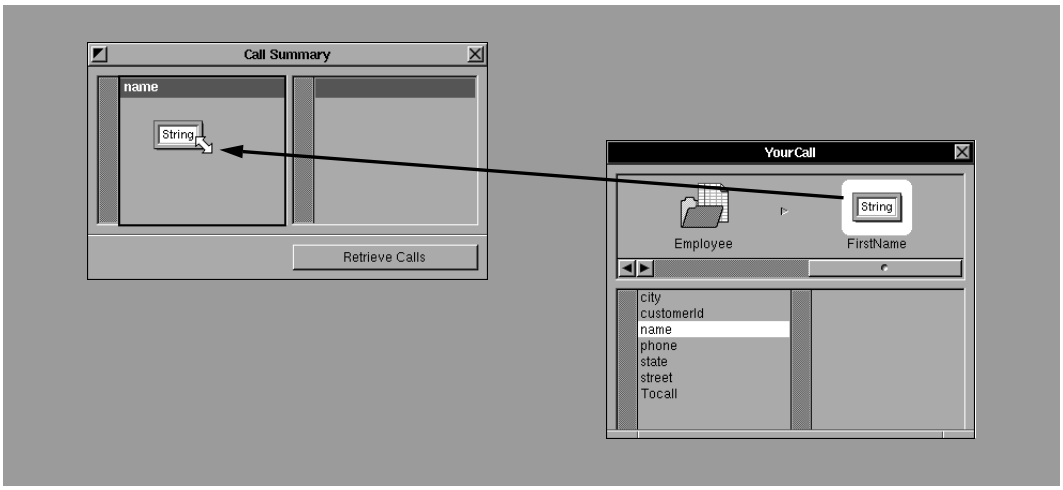
browser displays customer name fields, while the detail browser displays call topic fields. In addition, a Retrieve Calls button is placed in this Window.



Second, the Call Information window is redesigned slightly. A new text field is added for displaying and editing the Topic field. Just two buttons are included in the form, Clear Form and Save Call.

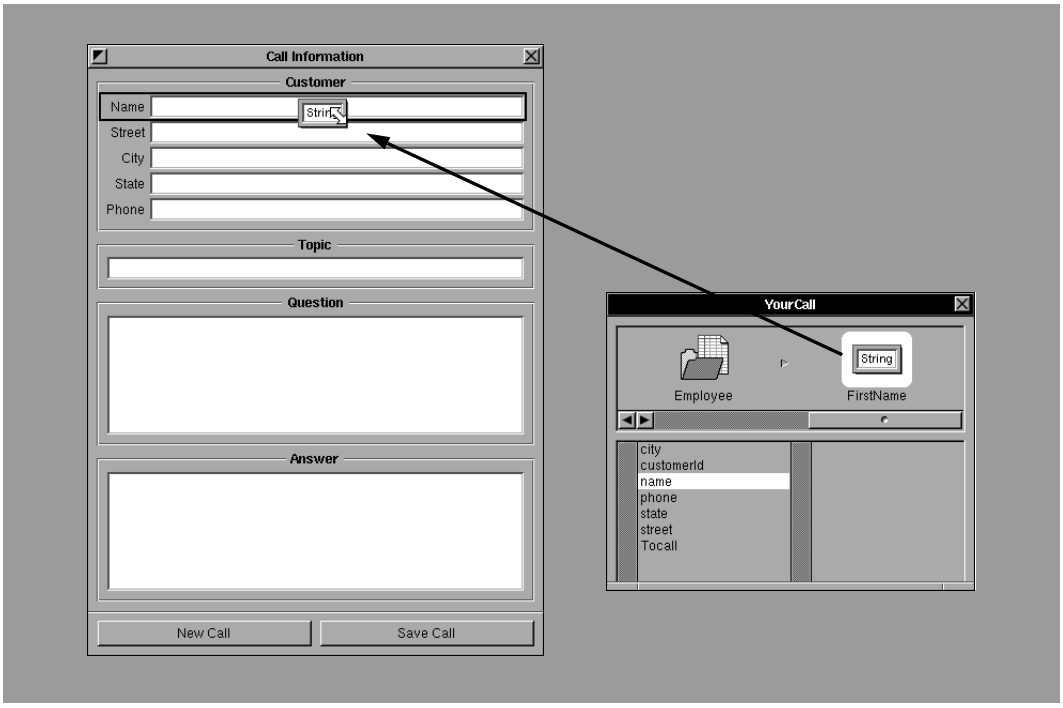


To connect the user interface to a database model, you drag a DBModule object from the Database Kit palette and drop it in the interface. You then choose the model to use, and a browser displays that model's entities, attributes, and relationships. From this browser, you drag and drop connections between the attributes in the model and objects in the user interface. For example, you can place customer names in the master browser in the Call Summary window:



Drag name from the model browser and drop it in the master DBTableView

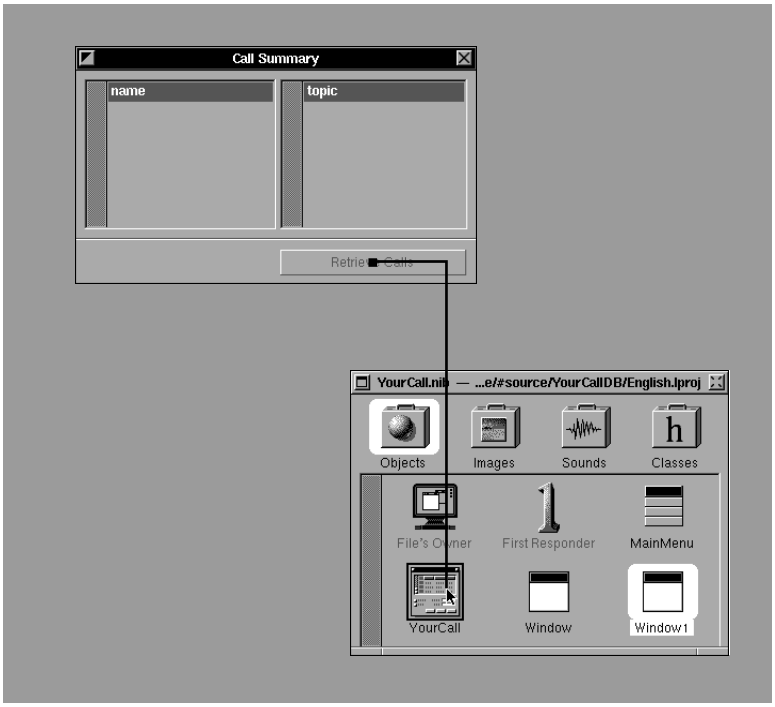
Note that the title of the column in the DBTableView changes to match the name of the attribute. You can also add attributes to the text fields in the form using the same drag-and-drop technique:



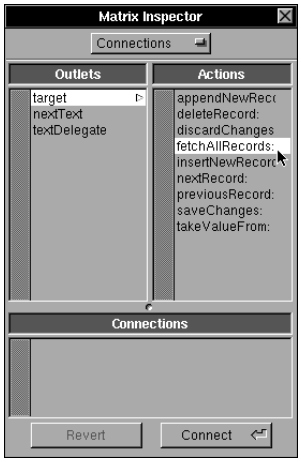
Drag name from the inspector and drop it in the Call Information Name field

Once the attributes in the model are connected to the appropriate fields in the user interface, you can connect the DBModule as the target of buttons in the interface. As a general purpose transfer object, DBModule offers a number of

action methods for use in standard database operations. For example, DBModule has a fetchAllRecords: method to use as the action of the Retrieve Calls button.

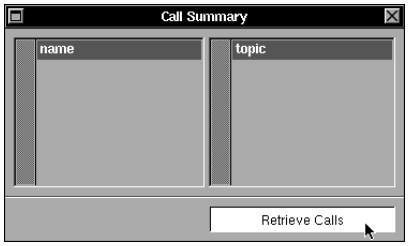


Control-drag from Retrieve Call to the YourCall DBModule

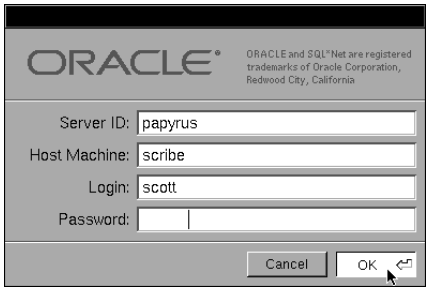


Double-click **fetchAllRecords:**

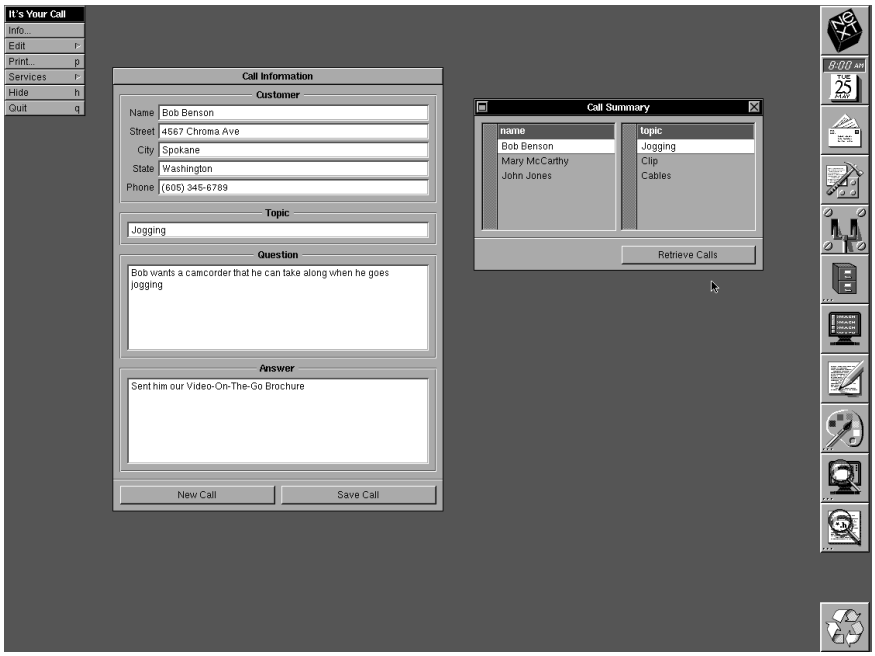
Because the model used by the DBModule is created directly from the database, Interface Builder can provide live interaction with that database once the user interface is completely connected. When you put Interface Builder in Test Mode, clicking a button connected to the DBModule initiates database access:



Click Retrieve Calls



Log in to the database



Once you're logged in, the actual records are displayed

Once the data is displayed in the form, you can use the various interface components to browse the data. For example, when you browse through topics, the data displayed in Call Information changes. When you browse to another customer, both the topic browser and the Call Information data change.

This example reiterates the power of Objective C's dynamic binding feature. A running application, Interface Builder, creates instances of Database Kit objects that know how to connect to and retrieve data from the database server. Those objects then communicate with objects in the user interface to display data from the server.

Database Kit: Summary

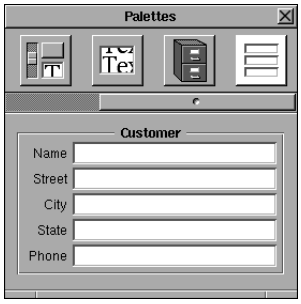
In redesigning YourCall to take advantage of the Database Kit, we created a powerful version of the application without a single line of code. This example demonstrates both the power of object-oriented programming and the flexibility of the Database Kit. Using this kit, you can create client-server applications even more quickly than with forms tools, while NeXTSTEP provides both interaction with other applications and the full programming capabilities of Objective C.

CREATING CUSTOM INTERFACE BUILDER PALETTES

You've seen how Interface Builder lets you add an Application Kit object to your application by dragging it from the Palettes window. Once in your application, such an object can be directly manipulated to resize or relocate it, or to edit its text. In addition, Interface Builder's Inspector lets you alter many of the attributes that can't be set directly—for example, how a button highlights, or whether a window is resizable.

In the Database Kit example, you saw how Interface Builder can load and use custom palettes such as the Database Kit palette. Through custom palettes, Interface Builder extends the techniques of direct access and manipulation to objects that weren't anticipated when Interface Builder was designed.

A custom palette is a palette that you or another developer creates; it's identical in operation to any of Interface Builder's standard palettes. A button at the top of the Palettes window gives access to a custom palette. For example, as a custom palette object, the Customer Form from YourCall looks like this:



With this custom palette, you can add one or more Customer Forms to an application in the same way you would add Application Kit objects. Once in an application window, the Form can be repositioned and resized using the mouse. Using the Connections Inspector, you can connect the Customer Form to other objects in the application.

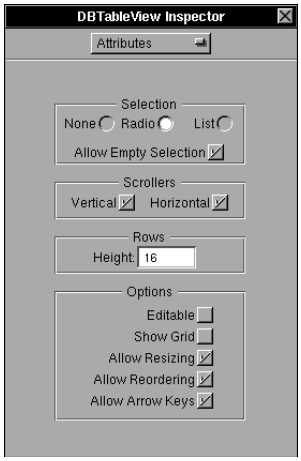
Creating a custom palette is simple; it's done with the same techniques as those used to create a standard application. For example, to create a custom palette containing a Customer Form, you start a new Project Builder project and specify its type as a Palette project. When the new project opens, you open the template user interface archive file for the project.

The template file contains one panel which holds the object or objects that will appear in the Palettes window. To create the interface for the Customer Form palette, you might repeat the steps in creating the original form. However, you can also simply open the interface file for YourCall, copy the Customer Form, then paste it into the palette panel.

To build the custom palette, you choose Build from the Project Builder interface. To place the new palette in Interface Builder, you choose Load Palette from the Tools menu.

The Customer Form example demonstrates a custom palette object created by simply grouping several standard Application Kit objects. More typically, however, you'd put objects you've designed on custom palettes to give all members of the development team access to those objects.

In some cases, you may want to provide an inspector for each class of object in your palette. Interface Builder lets you supply a custom inspector that offers text fields, popup lists, scrollers and other means to set attributes for objects in your custom palettes. For example, the Database Kit palette provides an inspector for setting number of rows, scroller configuration, and other attributes of the DBTableView.



In other cases, a palette object may be simple enough that no inspector is required. Or, you may find Interface Builder's standard inspectors sufficient for your needs. For example, since the Customer Form palette uses only standard Application Kit objects, there's no need to create an inspector. Interface Builder will display the correct inspector for each component object in this custom grouping.

Custom Palettes: Summary

In most programming environments, creating your own user interface devices is a difficult, time-consuming task. In NeXTSTEP, object-oriented programming helps you create custom user interface devices easily, either by collecting and customizing existing objects or by subclassing existing objects to give them just the new behavior you need. Once you've created useful objects, Interface Builder's custom palettes make it possible for anyone on your programming team to reuse them, by graphically adding them to any application where they may be needed.

USING INTER-APPLICATION SERVICES

In NeXTSTEP, an application can access services beyond those provided by its own code. You've already seen how the Header Viewer service is made available from Edit, the text editor that comes with NeXTSTEP. When you select some text in an Edit document and open Edit's Services menu, this is what you might see:

From this menu, the selection can be sent to Header Viewer for lookup in NeXTSTEP developer documentation or the NeXTSTEP header files. It can also be sent to the Digital Webster application to get the definition of a word or phrase, to Mail to be added to an electronic mail message, to the Workspace Manager to be opened as a file or directory, or to Quotations to be used to search for quotations containing the text. What makes this form of inter-application communication even more remarkable is that the client application (in this case Edit) gains access to these services without needing any prior knowledge of them. The items in the Services menu are added and updated by the NeXTSTEP environment.

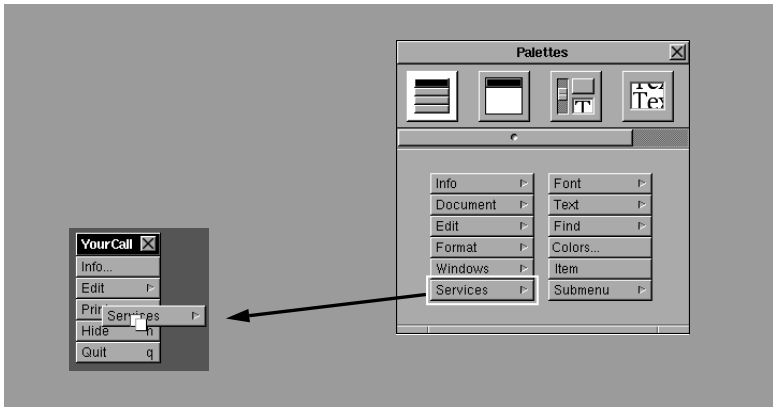
The Service Client

An application can become a client of the services system by:

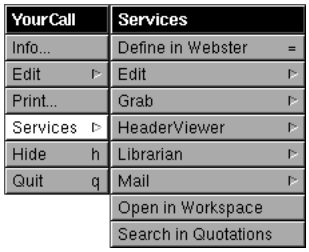
- Having a Services item in its menu.
- Having some object within the application register the data types it's willing to send to and receive from the service provider. This object must be able to send data of the specified format when asked to do so by the services system.

If your application contains a standard NeXTSTEP Text object, the second requirement is met automatically. A Text object registers the data types it can send and receive. But even if your application doesn't use a Text object, configuring another object to work with the services system is quite straightforward.

Since YourCall contains several Text objects for data entry and display, it needs only a Services menu item to participate in the services system.



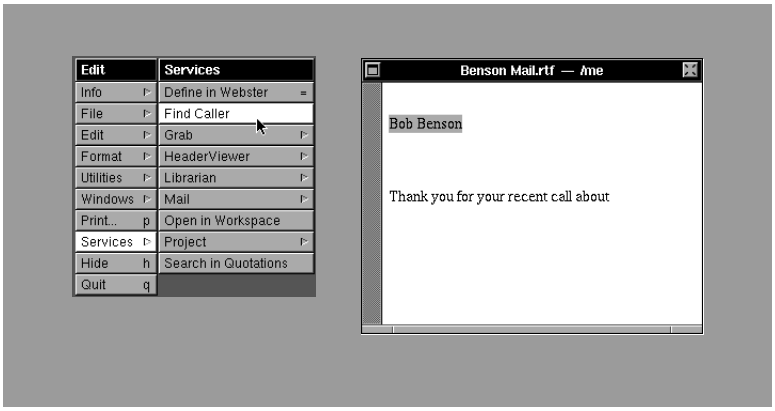
Once this menu item is added, the user can, for example, select the question or answer in the Call Information Window and send it as electronic mail using the service provided by the Mail application.



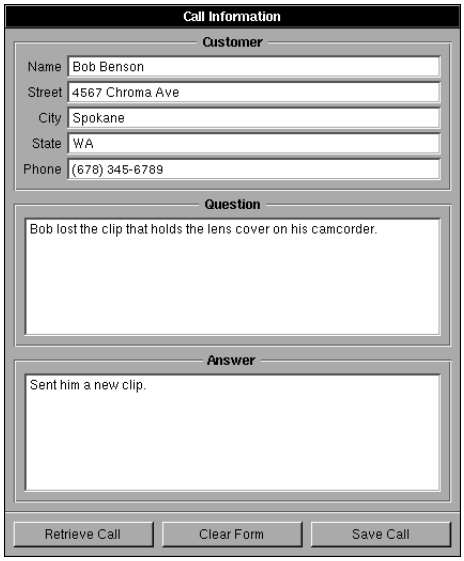
The Service Provider

In NeXTSTEP, an application can make its facilities available to other applications by advertising a service. The advertised service appears as a command in the Services menu of other applications. The Application Kit constructs the entries in an application's Services menu by matching the types of data it can send and receive with the types handled by the registered service providers. For example, if YourCall were configured as a service provider to look up names and retrieve records in the database, and Edit's Services menu would update to contain a command for doing this—say Find Caller.

From Edit—or from any other text editor or word processor—a customer support representative typing a letter would then be able to instantly access a specific caller through the Find Caller service:



When the support representative chose Find Caller in the Services menu, the YourCall application would automatically start, and the record for the customer would be retrieved and displayed.



The Services Table

An application advertises its service through a text table that's incorporated into its application bundle. This table lists the protocol that other applications must use in communicating with the service provider. For example, YourCall might have this table:

```
Message: lookupName
Port: YourCall
Executable: YourCall
Send Type: NXAsciiPboardType
Menu Item: Find Caller
```

The table specifies the *message* an application must send YourCall to use its service, the name of the inter-application communication *port* to be used for messages, the name of the application's *executable* file, the *data type* that YourCall expects to find on the pasteboard when it receives a request for its service, and the title of the *menu item* to be added to client applications' Services menus.

The Services Delegate

YourCall, as described so far, needs a few small modifications to handle service requests. At startup, it needs to identify an object that will handle service requests. In addition, that service handler needs to be able to respond to the message advertised in the table described above. Implementing these modifications requires the use of *delegates*.

A *delegate* is a kind of outlet—specifically, an object that acts on the behalf of another. As its name implies, the delegate shares responsibility with the object it is associated with. A number of Application Kit classes use delegates to let you synchronize the custom behavior of your application with standard NeXTSTEP behavior. In implementing a service provider such as YourCall, you need to identify two delegates, the *services delegate* and the *application delegate*.

The services delegate is an object within the service-providing application that's responsible for receiving service request messages. In YourCall, it's the object that will handle the lookupName message. You establish the services delegate by sending the message:

```
[[NXApp appListener] setServicesDelegate:self];
```

This line of code nests two message expressions. The first message goes to NXApp (a global variable that identifies an application's Application object) requesting its Listener object. A Listener object monitors the application's interapplication-communication port (the one advertised in the services table listed above) and relays service messages to the services delegate. The second message, setServicesDelegate:, goes to this Listener, asking it to establish self—the object sending the message—as the services delegate.

To process a services request, the services delegate needs to be established early, after the application starts, but before the user interface is displayed. The *application delegate* is provided to let your code respond to significant events in the lifetime of an application, such as starting, quitting, and becoming active or inactive. To make sure the service delegate registers its services early in the life of the application, it needs to become the application delegate and it needs to implement an appDidInit: method to actually register the service.

Subclassing CallController

The best way to implement a service provider version of YourCall is to create a subclass of CallController—CallService. CallService inherits all the behavior of CallController, and implements the two new methods required to act as a service provider. An instance of CallService replaces the CallController instance in YourCall's Interface Builder file.

To register itself as the services delegate, a CallService object needs an appDidInit: method. The Application object sends this message to its delegate just before the application begins accepting user-generated events. CallService uses the code above in its appDidInit: method to set itself as the service delegate.

Another new method, requestCall:userData:error:, is included in CallService to respond to a service request from another application. This method reads name data from the pasteboard, places that data in the name field in the customerForm, then sends a retrieveCall: message to self.

The code for CallService is listed and described in detail in the code listings at the end of this guide.

Inter-Application Services: Summary

All NeXTSTEP applications can participate in the Services system, both to take advantage of services provided by others and to offer services to others. NeXTSTEP provides simple hooks for taking advantage of this feature; connecting your applications to the NeXTSTEP Services framework requires just a few simple modifications to any application.

INTER-APPLICATION COMMUNICATION WITH DISTRIBUTED OBJECTS

Many kinds of mission-critical applications need to be able to share data or functionality, especially those deployed in large, networked office environments. In the last section, you saw how NeXTSTEP applications can share services among one another. At the beginning of this section, you saw how to use the Database Kit to create client applications that access and share data in a central database server application. These represent the two ends of the spectrum of interapplication communication.

Many mission-critical applications may benefit from functionality somewhere between these two. You may need to create several applications that work together more directly than through NeXTSTEP's Services feature. Or you may need to create client-server applications that operate outside the realm of commercial database management systems. In any case, NeXTSTEP Distributed Objects provide the answer—a simple, object-oriented way to implement customized communications between applications.

Distributed Objects

Using Distributed Objects, applications provide services to clients (or between peers) by vending objects. Once an application starts and registers a server object, other applications can request connections to the server, in essence creating local versions of that server object. A client application uses the local proxy object to communicate with the server object, sending messages to its proxy as though it were the server object.

Note that while client-server terminology is used to describe this relationship, applications may establish any number of communications links using Distributed Objects, with no constraints as to which is the server and which are the clients for any given link. Thus peer applications may interact through a central server application, through networks of client-server relationships, or through some combination of models.

Protocols

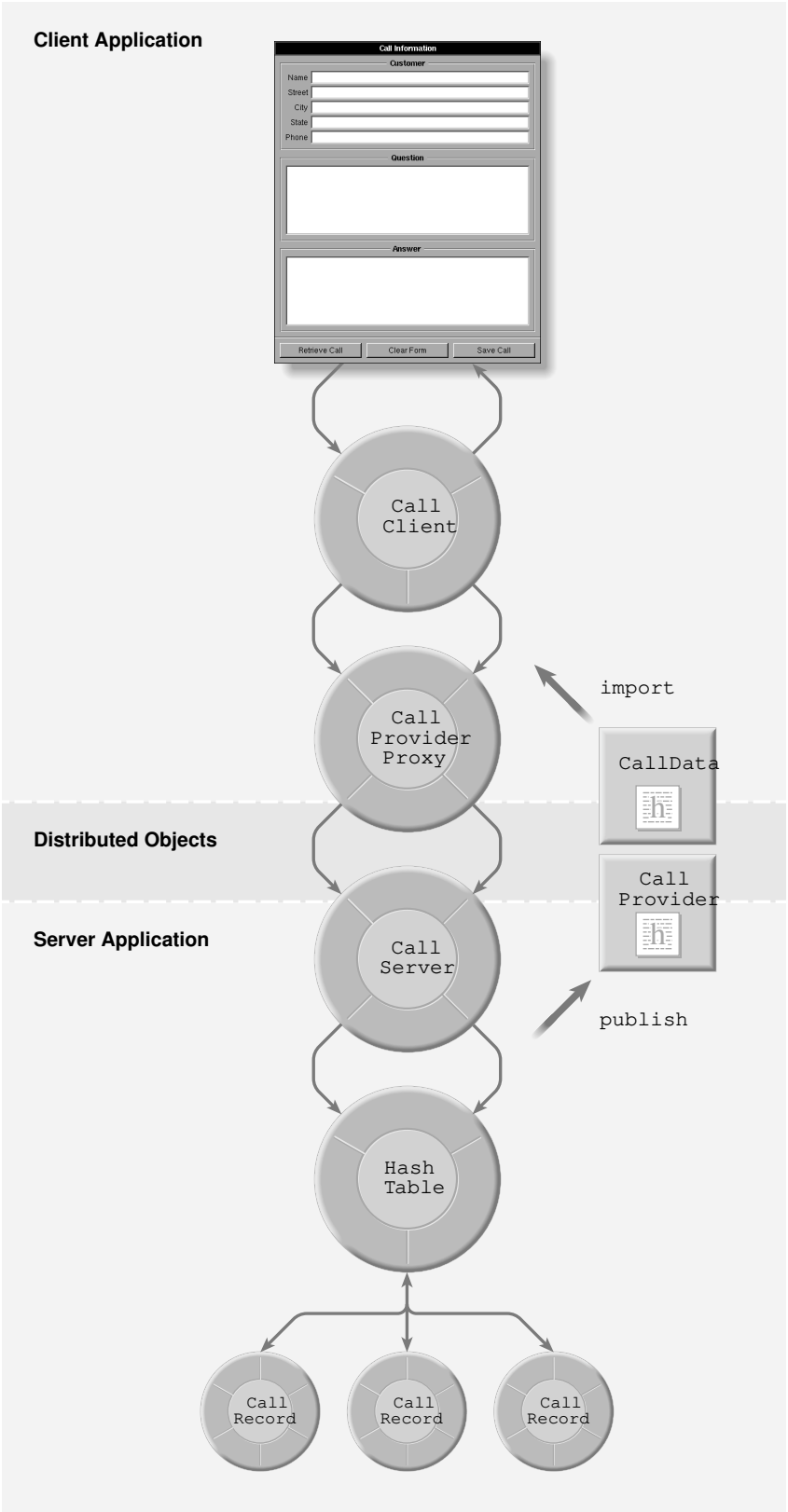
To use Distributed Objects, an application needs to know the messages it can send to the remote object through its local proxy. The standard way to make an object's methods available to others is through the class interface (.h) file. However, most applications are created and compiled as separate, stand-alone applications. An organization implementing a server application may not want to make all the server object's methods and instance variables public to developers of client applications. Instead, they may want to publish a short list of methods specifically intended for client interaction. In addition, a server may want to receive objects from a client that respond to a particular set of methods. Thus it's useful to be able to specify certain methods that an object must implement without fully specifying the object.

Objective C provides a mechanism, *protocols*, by which such method declarations can be made public. In the example that follows, you'll see how protocols can be used to specify methods for client-server interaction.

Designing YourCallClient and YourCallServer

The original YourCall application is an ideal candidate for redesign with distributed objects. Any number of customer support representatives might need to access call data at the same time. Each of them would run the client application on their desktop computer. The server application could be run on any NeXTSTEP system in the office, since Distributed Objects are fully network capable.

As a client-server pair, the original application divides logically in two, as shown in the following diagram:


Client Application
Call Information
Customer
Name
Street
City
State
Phone
Question
Answer
Retrieve Call
Clear Form
Save Call
Call Client
Call Provider Proxy
import
CallData
Distributed Objects
Call Provider
Server Application
Call Server
publish
Hash Table
Call Record
Call Record
Call Record

YourCallServer implements the parts of CallController that set up and interact with the HashTable database. The class created to fill this role is called CallServer—a subclass of CallController. The server also implements the CallRecord class, much as in the original YourCall application. Both CallServer and CallRecord remain hidden from the client application. In their place, the server application publishes two protocols, CallProvider and CallData.

The CallData protocol specifies all the methods for accessing the data for a call—name, setName and similar methods. CallRecord conforms to the CallData protocol. The CallProvider protocol declares three methods for accessing the service: lookupCall:, newRecord, and storeCall:. lookupCall: is specified to accept a character string representing a name from the client-side and return an object conforming to the CallData protocol to the client. newRecord is specified to provide an object conforming to the CallData protocol for the client to fill with call data. storeCall: is specified to accept an object conforming to the CallData protocol from the client for storage by the server. CallServer conforms to the CallProvider protocol.

YourCallClient implements the user interface and establishes a connection to the server. The class created to transfer data between the user interface and the server is called CallClient. It keeps all the methods and most of the instance variables of the original CallController. However, instead of having a callTable instance variable, it has a callProvider instance variable to refer to the proxy for the server object implementing the CallProvider protocol. Another change is in the code for the methods that save and retrieve records; these methods interact with the proxy rather than the HashTable.

The Protocols

The CallData protocol is published by YourCallServer and the server's CallRecord class conforms to this protocol. It specifies all the methods for accessing the data in a call:

```
@protocol CallData
- (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;
@end
```

Note that the argument and return types in the protocol are exactly the same as originally specified for CallRecord. This protocol is included in the source

directories for both client and server, in the file calldata.h. In its version of the file CallRecord.h, YourCallServer uses the code in bold to conform to the CallData protocol:

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

@interface CallRecord:Object <CallData>
{
char *name;
char *street;
char *city;
char *state;
char *phone;
char *question;
char *answer;
}

- read:(NXTypedStream *)theStream;
- write:(NXTypedStream *)theStream;
- free;

@end
```

The methods in the CallData protocol no longer need to be declared in the CallRecord interface (.h) file. They're assumed to be implemented because of the <CallData> declaration in the interface line.

The CallProvider protocol is specified as:

```
#import "callData.h"

@protocol CallProvider
- (id <CallData>)lookupCall:(char *)fetchName;
- (id <CallData>)newRecord;
- storeCall:(id <CallData>)theRecord;
@end
```

The type declaration id <CallData> specifies that the lookupCall: and newRecord methods return objects conforming to the CallData protocol; similarly, storeCall: accepts an object conforming to this protocol. This declaration is included in the source directories for both applications, in a file callProvider.h.

Retrieving a Record

When the YourCallServer application is started, its CallServer object initializes the HashTable containing the CallRecords, then identifies itself as the object being vended by the application.

When the user starts the YourCallClient application, its CallClient object requests a proxy for a CallProvider object, with the code:

```
- init
{
    [super init];
    callProvider =
        [NXConnection connectToName:"CallDataServer"];
    return self;
}
```

Then, as the user enters data in the form and manipulates the buttons, the client interacts with the server. For example, to implement record retrieval, CallClient's retrieveCall: method is modified slightly from that of CallController, as indicated in bold type.

```
- retrieveCall:sender
{
    const char *fetchName;
    id <CallData> fetchRecord = nil;

    fetchName = [customerForm stringValueAt:0]; // name
    if (fetchName && strlen(fetchName)) {
        fetchRecord = [callProvider lookupCall:fetchName]
        if (fetchRecord){

        /*... display the data in the form ... */

      }
      else {
          /* attention panel */
      }
    }
    else {
        /* attention panel */
    }
    return self;
}
```

The change to the code for looking up a record on the client side is very simple. Rather than requesting the data from the HashTable, it requests it from the callProvider proxy. The CallServer method invoked in this line, lookupCall:, is a simple one-liner:

```
- (id <CallData>)lookupCall:(char *)fetchName
{
    return [callTable valueForKey:fetchName];
}
```

As you can see, this server method simply uses the HashTable valueForKey: method to find the record requested by the client. It uses a standard C return statement to pass the retrieved object back to the client. You'll find more complete listings of the client-server code in the code listings at the end of this guide.

Distributed Objects: Summary

As this example demonstrates, Distributed Objects make it easy to implement interoperating applications. Messaging between applications is exactly like messaging between objects within an application. This example also demonstrates how protocols let servers and clients coordinate their behavior while hiding implementation details.

SUMMARY

This section has demonstrated a number of ways in which NeXTSTEP can be used to enhance your mission-critical custom applications, improve access to data, provide communication between applications, and make the code you've already written and debugged easily reusable.

The rest of this guide provides a deeper discussion of object-oriented programming and NeXTSTEP, revealing the framework they provide for applications like YourCall—and for the more sophisticated applications that you will write.