Chapter 5: Messaging and Dynamic Binding

mes•sage *\'mes-ij\ n*
[ME, fr. OF, fr. ML missaticum, fr. L missus, pp. of mittere]
(13c)
1: a communication in writing, in speech, or by signals
2: a messenger's errand or function
3: an underlying theme or idea
Webster's Ninth New Collegiate Dictionary

Now that you have an idea how easy it is to create objects, our next consideration is the ways we communicate between objects. This is called messaging. Objects communicate in a manner very similar to the way people in an office communicate - using messages written on notes. If I was to send you a note I might put who the note was to, the contents of my message and any additional information at the end. In object based computing environments, any object can send a message to any other object. The message contains who the message is for, what the message content is and any additional information required. In object based computing, the message content is always the name of a method of the **target** object. An example is listed in figure 8.

Sue, Let’s do lunch Time: 1200 noon From: Betty	[sue doLunchAt:1200 with:betty];

Figure 5-1: Written note (left) compared to a message statement.

The sample on the left is a sample note you might leave in a person’s mail-box. It contains items like who the message is to (Sue), what action is being requested (meeting for lunch) and the arguments (meet at noon and who the note is from). The message statement that would correspond to this statement is very similar. The example on the right contains the message statement:

```
[sue doLunchAt:1200 with:betty];
```

Note that the message statement is traditionally enclosed in square brackets and ends with a semicolon. The target of the message is always the first argument. In this case the target object or the destination object is called "sue". The second component is the name of the method that we wish sue execute. In this case the actual name of the message is "`doLunchAt:with:`". Note that the colons are actually part of the name. This tells the compilers to check for arguments. In this way we can assure that the correct number of arguments are being sent ,of the correct type, and in the correct order. The compilers carefully cross check the types (integer, floating point or stings) of each argument when the program compiles so that by the time the program runs we are assured of getting correct results.

A good demonstration of messaging can be seen directly within the Interface Builder. To see this, start up Interface Builder and create a new application. Drag a slider object and a text object onto the main window. While the control key is being pressed, drag a connection from the slider to the text object. You should see a black "rubber band" line appear as in figure 9.

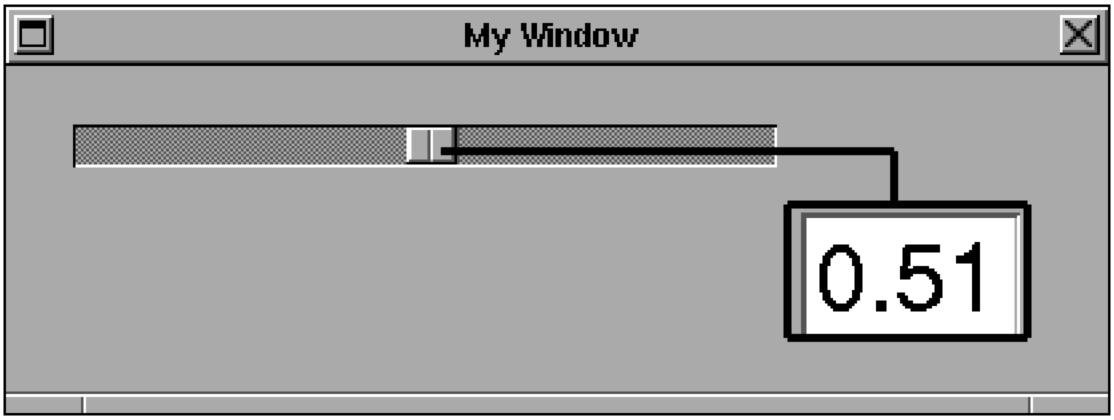

Figure 5-2: sending a "takeFloatValue:" message from a Slider to a TextCell object

In this case the slider is the source of the message and the textCell is the target. After you release the mouse button over the text object you will see the inspector appear in the lower right corner of the screen. It will display a list of methods that the textCell can respond to. In this case since the slider sends a floating point number we want to select the "takeFloatValue:" from the right column of the inspector and then select the "Connect" button in the lower right hand corner of the inspector window. A small knob should now appear next to the selected method indicating that the connection has been made. You can now test your connection by selecting the "Test Interface" option from the File main menu. This simulates the user actually running the program interface that you have just created. After all the Interface Builder windows disappear you will see only your program's main menu and window. By moving the slider you will now begin sending messages to the textCell object.

The message statement is very powerful: it is the only statement that we need to add to a language to make it "objective". These statements have been added to C, FORTRAN[1] and there is justification to creating messaging extensions to languages such as C++, LISP, PROLOG, Pascal, DSP Assembler and even COBOL. By adding messaging to these languages, developers could retain their inventment in their current software base and still provide a NextStep interface to their older programs. Adding message statements to a language can be done in a "pre-processor" phase that translates the message statements into an equivalent library call that the native language compiler can recognize. The important thing to realize is that the message statement is language independent. You simply pick the language that best represents your problem and add messaging to it.

Benefits of Messaging

Messaging is very similar to a subroutine call in some ways. It is usually one line of code that causes a larger and possibly remote section of a program to be executed. Both subroutine calls and messages have arguments and these arguments each have data types that must be checked at some point. But message statements are much more flexible in many ways. We

[1] The Absoft FORTRAN compiler was available for the NeXT system even before release 1.0

can send a message statement off to the object but that object does not have to handel that message by itself. If it does not have the named method in its list of messages, it re-directs the message to its super-class. If the super-class does not respond it also sends the message on up the inheritance tree structure. In traditional programming languages when a subroutine is inserted into a program the program flow has only one choice. The program control must go to a subroutine in one known location.

Searching for Methods

Messaging brings an additional level of flexibility to our environment. Just like DOS or UNIX allows us to include our own "shell" commands in the search path, messaging allows us to search for method names in a path. In most cases the search path only goes up the class hierarchy toward the root object. This path can be changed at any time by a technique know as "delegation"[2]. Delegation allows objects to delegate messages they receive to other objects even in unrelated hierarchies.

Because we have a search path for messages we have the ability to intercept messages. We can change the functionality of a message completely or we can just change one small aspect of it and then send the message on. For example if we wanted to change the way a slider was drawn, perhaps we wanted to just add a page or record number to the knob of a slider. That could be done by sub-classing the slider object and just changing the methods that draw the slider. We do not have to change the way the slider sends and receives messages or the way the initial values are created. We can easily change only the attributes that we are concerned with. The bottom line that is it is very easy to customize objects to meet application specific needs without a lot of time or code.

Polymorphism

One additional feature of messaging is the ability for different objects to receive the same message and execute different methods. My favorite demonstration of this is using the NeXT mail system. If you attempt to send a message you can type text into the send window and copy, cut and paste the text using selection from the main Edit menu. If you click the pair of lips and record a sound into the "Lip Service(TM)" utility (you will need a microphone to do this) you will find that you can select the edit button inside the "Lip Service" area and use the same copy, cut and paste messages form the main Edit menu to copy, cut and paste sound as well as text. This is illustrated in figure 10.

[2] See Chapter 6 of the NeXT System Reference Manual for more details about delegation.

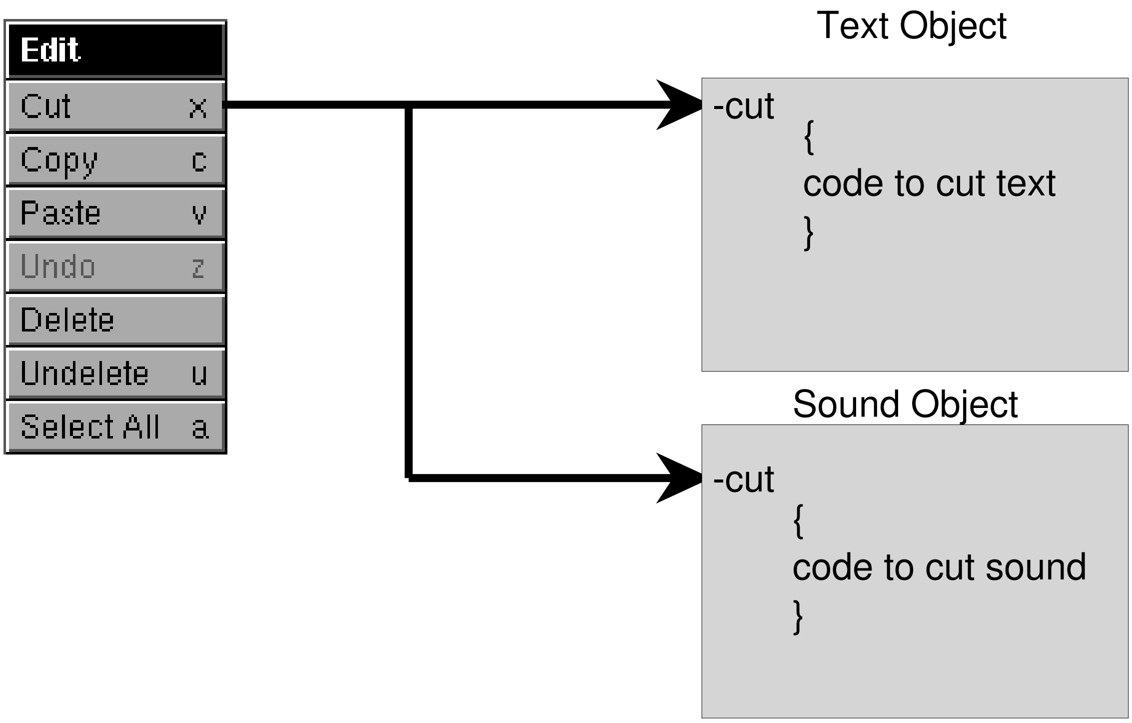

Figure 5-3: Demonstration of Polymorphism within NeXT Mail.

(Things to do: add pictures of text and sound objects to the above figure)

Although this is a fun example to demonstrate to others, the underlying principals should not be overlooked. In this example we are sending the same messages to two different objects: a text object and a sound object. Each of these objects has completely different types of internal data and each object must have different copy, cut and paste methods to perform the tasks on their own internal data.

Polymorphism means being able to assume different forms. The roots literally are *poly* which means many,and *morph* which means form. In the programming sense, the message causes different sections of code to be executed. The messages in a sense "turns into" different types of code that is executed. For application developers polymorphism means that if our applications need to edit objects, we simply add copy, cut and paste methods to our objects. We can use the same edit menu that is used to edit text and sound as well as our own custom objects. This saves the developers time and provides the user with a more consistant user interface.

Message Protocols

Messages can be grouped together. A group of messages that is related and often used together is known as a protocol. The messages commonly grouped together under the main edit menu such as cut, copy, paste, undo, delete, un-delete and select all define an edit protocol. Protocols are important not only for consistency between applications but also for reasons of extendibility and maintainability. When we have full motion video objects in future computers we will be able to add video editors to our mail systems simply by using the same edit protocols.

One example of how polymorphism allows programs to be easily maintained is in the Draw program that is supplied with every NeXT system. You will find that there is a protocol that has been established for all objects that the editor can manipulate. You must be able to create objects, read the objects off the disk, draw the objects on the screen and be able to write the objects back onto disk. If you wanted to be able to edit some other objects, such as pie charts, bar charts or X-Y plots of data they could easily be added to the Draw program without changing the original source! You simply add the files to the Draw directory and implement the create, draw, read and write methods and you have added a new object to your draw program. Once again we see that we can easily customize programs by using object oriented programming techniques. By extending programs like Draw, users don't have to start from scratch and the total amount of new code they must maintain is much less than if they started from scratch.

The word protocol is very familiar to people who create data communication networks. This should come as no surprise since that is essentially what we are doing with objects. We are creating a networking mechanism for them to exchange messages. Although we usually think of messages in a Objective C program only communicating with other objects within that program there are many ways for messages to flow between programs. These programs can be on different processors with different instruction sets on local as well as remote networks.

Dynamic Binding

When we decide that we want to send a message from one object to another, we must specify a destination for the message. There are two times in the compilation stage that this can happen: when the program is compiled form its source to object code or after the program is run. Most traditional compilers make an association between a subroutine and where that subroutine location is when the program is compiled. With Objective C we have a slightly more rich environment to chose when the destination is specified. We delay the decision till the program is actually running. This is called late binding 0r dynamic binding. Dynamic binding allows the Interface Builder to have a "Test Interface" menu. This allows user to test the objects within Interface Builder and their connections without ever having to compile or link their programs. Dynamic binding allows a greate deal of flexibility to the person who wants to customize an existing binary application. The traditional system integrator does not hav e the ability to add and substract modules from an existing program while it is running.

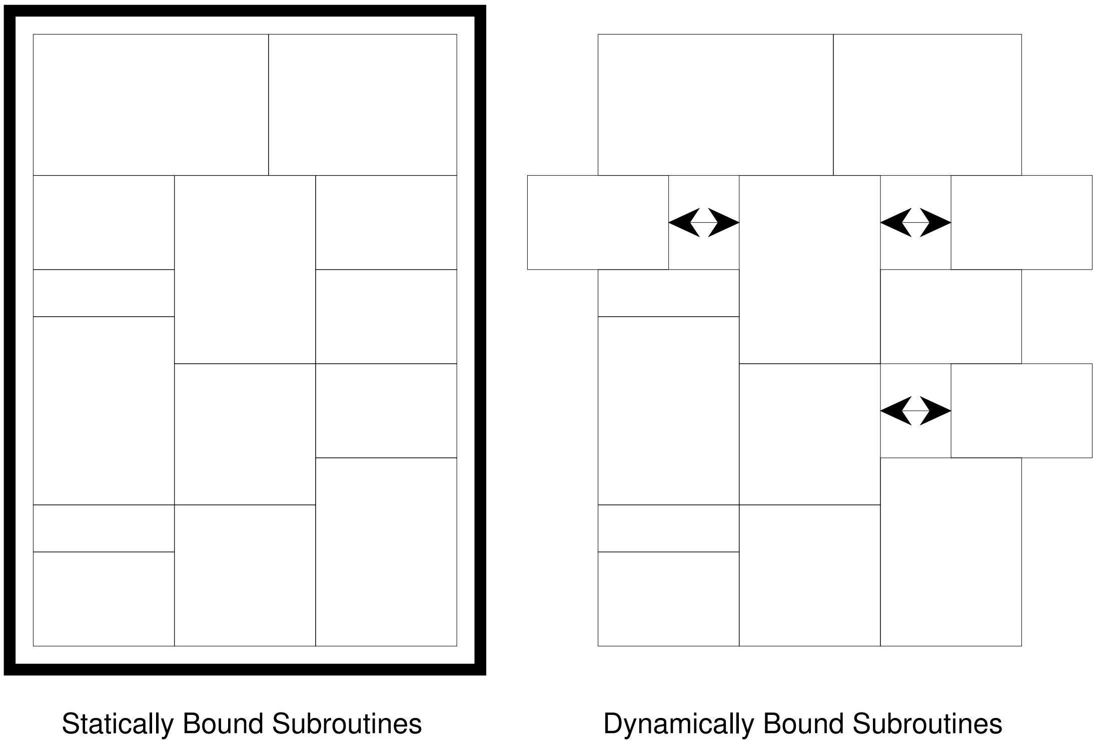

Figure 5-4: Static vs. Dynamic Binding

Future versions of NextStep may also support the ability to dynamically add existing modules to programs. This feature is known as "dynamic loading". It would allow future versions of the Interface Builder to have a "Load Palette..." menu option which will significantly ease the ability for groups of developers to share objects. Users will be able to load new objects into the Interface Builder as well as test the objects with the "Test Interface..." option.

Exercises

1) What are some of the messages used by other objects in the Interface Builder? What are some of the messages that windows respond to? How might you create a help panel by adding text to the a panel and then sending a "orderFront:" message from a menu cell to that panel? What happens if you add a "Print" menu and send the "printPScode:" message to a window, a View or any sub-class of View?

2) What if you were a dictionary object? What types of messages might you respond to? What might the arguments be?

3) What are some common protocols that you use every day? If you were attempting to create an object that drove a car what might your protocols be to drive the car? What are some messages that some cars might respond to but others might not?