

1. Introduction: Why Do I Need Prograph?

Overview

Prograph represents an entirely new approach to programming using a visual graphical representation of program code instead of source code text to build programs easier and more efficiently. This chapter will introduce the reader to the Prograph programming language and programming environment, and compare Prograph to traditional programming languages. Since Prograph is so different from conventional programming languages, we'll present a complete example program to demonstrate some of the revolutionary features of visual programming and Prograph. These features and more will be discussed in detail in later chapters.

Conventional High-Level Computer Languages

In the early days of computers, programs had to be painstakingly written as a series of ones and zeros. To simplify the task of computer programming, textual computer languages were developed that used words and textual symbols to represent instructions for the computer to carry out. These textual languages mirrored the text-based computer environments on which they were used, such as the UNIX command-line interface or DOS character-based screens. Several so-called “*high-level*” languages have emerged in the past few decades, resulting in a veritable “Tower of Babel” of computer languages -- C, Pascal, Forth, BASIC, Prolog, LISP, Ada, C++, SmallTalk and COBOL to name but a few.

With these languages, writing computer programs starts with planning a fixed order of program instructions. This sequence is sometimes represented as a graphical *flow chart* that shows where the program (or a section of it) starts and stops, when data is input by the user or displayed to the user, the sets of calculations carried out, and the logical decisions that the computer must make. The flow chart is designed to be easily understood, so that the programmer can grasp the program's structure at a glance.

Unfortunately, after outlining his or her ideas in this easily understood pictorial format, the programmer must then translate the sequence of instructions captured in the flow chart to the textual style demanded by a given programming language. Textual computer languages each have a unique rigid syntax for writing computer instructions, just as human spoken languages have their own rigid rules for constructing sentences. What is more important, the product of this program translation -- the textual source code (a sample of which is shown in Figure 1.1) -- is now *harder* to understand than is the graphical flowchart! For example, the reader of the code must examine it carefully to see which text corresponds to an operation, a variable or a constant, as well as which data is input to or output from the code.

So in writing the program, we've progressed from an easily understood graphical representation of the program (its flowchart) to a *less understandable* collection of words and symbols. And to top it off, we've spent *extra time and effort* to do this!!

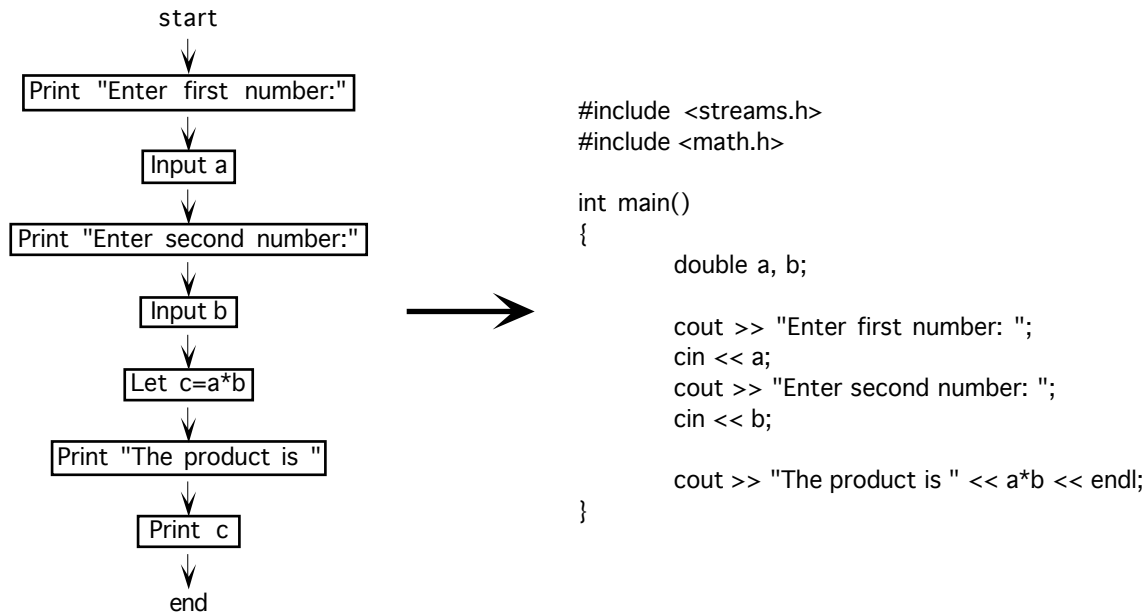


Figure 1.1: Comparison of a flow chart diagram to its equivalent C++ textual source code

To make things worse, computer languages are even more rigidly structured than spoken languages. In an English sentence, for example, you could accidentally leave out a comma or period or forget to capitalize a word, yet the reader would still be able to understand the meaning of the sentence. If a computer programmer were to forget to include a special symbol such as a semicolon or accidentally misspelled an instruction's name in a line of computer code, the interpreter or compiler "reading" the program would be much less forgiving. It may either choose not to execute the program at all or might run it incorrectly, freezing up or crashing the computer.

What is Prograph?

Programming would be much easier if we could just minimize the need for using text-based computer language syntax. This is where visual programming, and *Prograph CPX™* in particular, comes in.

The Prograph programming language is a revolutionary departure from traditional computer languages that uses little text. Prograph is a *visual (graphical) language* in which program operations and elements are represented not as textual symbols but as *pictures* or *icons* (see Figure 1.2), much like the way a graphical user interface represents disk directories and files with pictures instead of words. It becomes much easier to tell apart the data from the operations at a glance.

Graphical user interfaces tend to make computer users more productive. Likewise, a graphical programming language makes programming more efficient. Prograph programs may be built with a minimum of coding -- *a program written in Prograph typically takes about one-third to one-half of the programming time needed to write an equivalent program in a textual language like C*. This makes Prograph an ideal language for *rapid prototyping* of applications. This ease of use and efficiency doesn't sacrifice execution speed in the completed program. If you need a little extra speed for time-critical portions of the program, you can add external code written in lower-level languages like C at compile time.

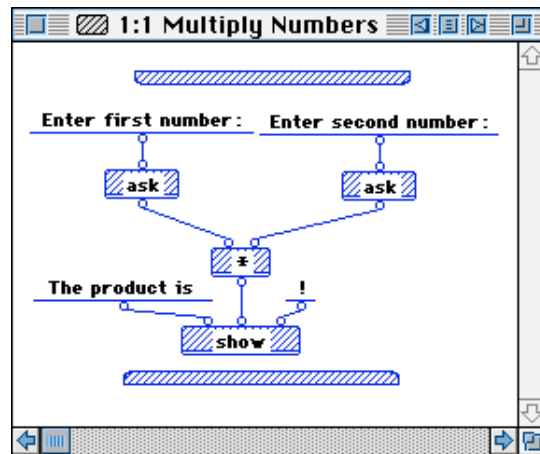


Figure 1.2: Iconic dataflow programming with pictorial representations of instructions

The efficiency of graphical programming has resulted in a number of high-level iconic programming tools tailored to create specific types of programs for specific types of tasks. For example, the LabView™ language is specialized for creating scientific and engineering programs for the laboratory or industrial setting. Unlike these other visual languages, Prograph CPX was designed first and foremost as a *general programming language* to build *any kind of program* -- graphical, word processing, scientific, business, musical, multimedia, communications, and so on.

Prograph, like LabView, is not only visual in nature but is also a *dataflow language*. Unlike textual programming languages, in which program control is laid out in a rigid order, much like people read a book -- from top to bottom -- in dataflow languages the program is represented as operations interconnected by links through which data "flows." In other words, while textual languages are *control-driven*, or executed in the order in which instructions are written, Prograph is *data-driven*, -- instructions are executed when all of their input data is made available. This means that computations and other operations need not be carried out in a *fixed* order. For example, in Figure 1.2, the "*" or multiplication command can execute *only* when "*" has received the data passed out of *both* ask's output data links towards "*". On the other hand, the exact order of execution of the two ask commands themselves is less important, and they may be

executed in any order (although we have the option of forcing them to operate in a fixed order if we wish).

Programming in Prograph means “wiring together” elements that receive and send data. Data flows into an operation, which acts upon it, then flows out of the operation and on towards other operations along data links. Which operation executes next depends upon when each operation’s data is made available to it. The ability to execute operations in a less order-independent fashion will allow Prograph programs to take full advantage of technology that will become more prevalent in the near future, such as parallel processing computers. In Figure 1.2, for example, separate parallel processors could execute each of the “ask” instructions *at the same time*, speeding up program execution even more.

Prograph was developed in the 1980's by the *Gunakara Sun Systems* (now *Prograph International*) in Halifax, Nova Scotia. The current generation of Prograph -- Prograph CPX -- includes such advanced features as a comprehensive code library that provides a ready-to-use relational data base and object-oriented user interface/application framework. But Prograph CPX is more than just a computer language and code library. It is also an *extremely well integrated program development system* that includes interacting graphical *program editor*, code *interpreter*, graphical *debugger*, and graphical *user interface builder*. The Prograph programming environment allows you to plan the computer program, write it, test it, and design its user interface -- all with a *single* tool. The debugger is an integral part of the program editor, so programs may be interactively written and tested piece by piece. You can even add new parts to your programs as you are running and testing them. After the program has been fully tested, Prograph’s code compiler will produce a fast stand-alone version of your program.

Prograph also allows the programmer considerable flexibility in the choice of programming style. Prograph supports the more traditional *structural procedural programming* mode, as well as the newer *object-oriented programming* style. This book will teach you how to write programs with Prograph using *both* programming philosophies, since both programming styles have their uses for different types of programs or even different parts of a single program.

Creating Prograph programs

Let’s dive right in and examine an example program in Prograph so we can see some of the key features of Prograph. We’ll make a simple address book program that will maintain a list of contact addresses, allow the addresses to be viewed one at a time, save the address list to a disk file and print each address. What’s extraordinary about this program is that we will implement it by writing only 14 routines -- even though the program includes a full user interface.

A Prograph program is contained within a *project*, which is subdivided into one or more independent modules of code and data called *sections*. Note that each section in the Section Window (see Figure 1.3) has three icons. They represent the three components

of sections -- classes, universal methods and persistents. These three components roughly correspond to object-oriented programming code, procedural code and global data, respectively. Sections may be *reused* as needed in other programs. In fact, our example program will be composed entirely of sections from the object-oriented Prograph *Application Builder Classes* framework that we'll reuse to manage the program's user interface.

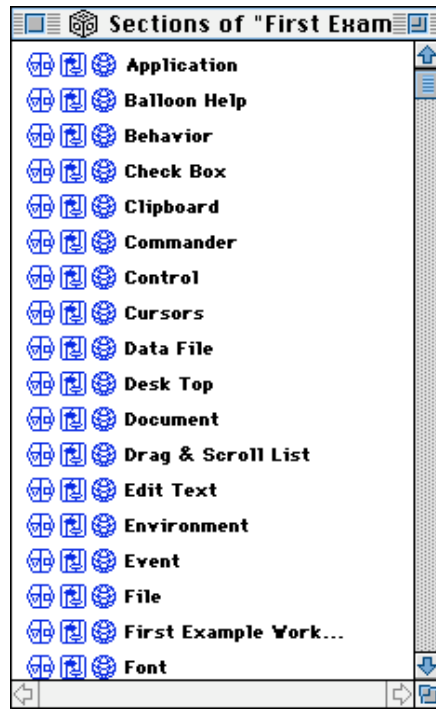


Figure 1.3: Sections window for the Address List program showing sections of the Application Builder Classes framework

We'll add the 14 application-specific routines that we'll write to one of these sections -- the one entitled First Example Workspace. This section is made to contain self-contained units of application-specific user interface code. Specifically, we'll add new code to the existing object-oriented code or *classes* of this section (see Figure 1.4)

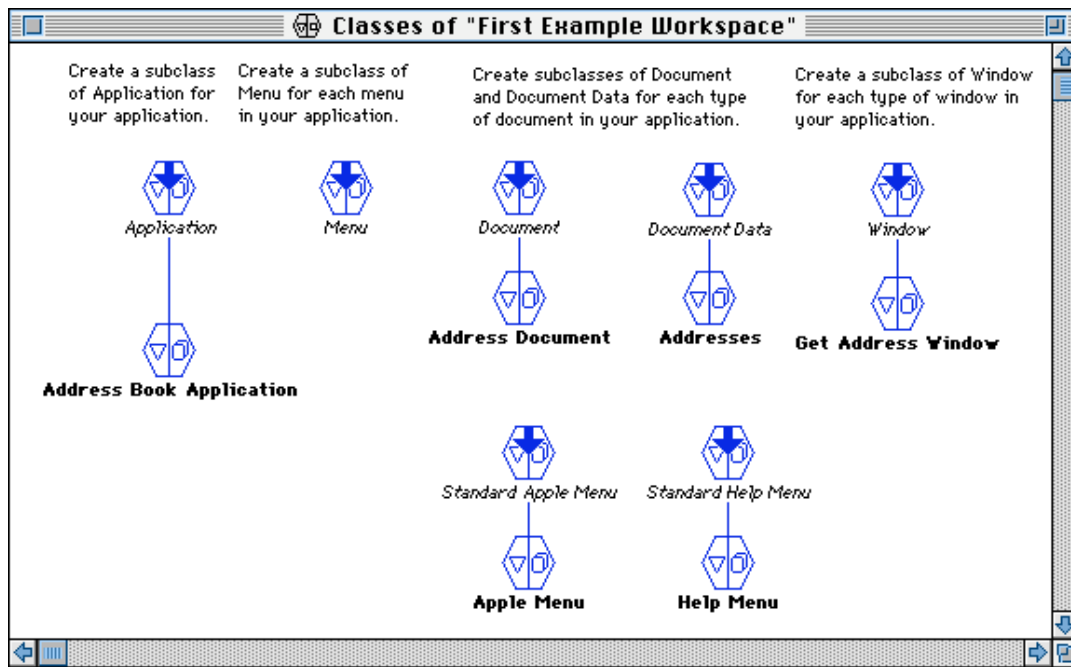


Figure 1.4: Classes of the First Example Workspace section

The majority of the icons in the First Example Workspace section depict some of the classes, or packages of code plus data, that have already been supplied with the *Application Builder Classes* framework, including the **Application**, **Menu**, **Document**, **Document Data**, **Window** and **Standard Menu** classes. Each class contains general code to manage one aspect of a user interface (in conjunction with code within other sections of the framework); for example, the **Menu** class manages user mouse selections of items in each of a program's menus and enabling/disabling menu items, while the **Window** class manages mouse selections within a window, as well as window opening, closing, activation, drawing, etc. We'll simply use these pre-existing code modules in our program to control our program's user interface for us.

Four of the classes in this window are special because they contain the application-specific code for our program. These are the **Address Book Application**, **Address Document**, **Addresses** and **Get Address Window** classes. How do we create these new classes? We don't -- we let Prograph create them for us.

One revolutionary feature of Prograph is how well integrated its application development environment is. The program editor, code interpreter and user interface design tools all cooperate fully with each other. This degree of integration helps us to build this program with a minimum of coding.

Prograph programs are built by first designing the program, then filling in its details with code. Let's begin by building the user interface of the program. By selecting the Edit Application menu item in the Prograph environment, we enter a series of

graphical application user interface design tools called the Application Builder Editors. The first of these is the Application Editor (Figure 1.5).

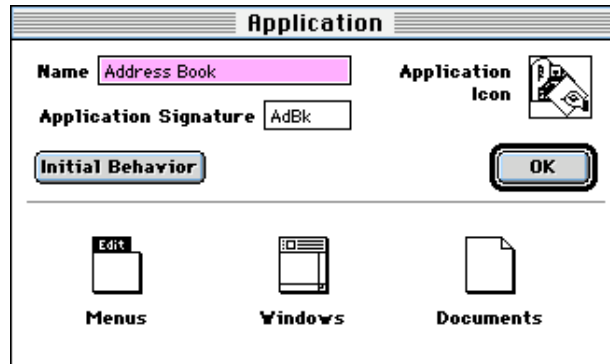


Figure 1.5: The Application Editor

We've given our program the name Address Book. Its four-letter program type (for the Macintosh computer) is 'AdBk.' By selecting one of the three icons on the bottom of the editor window, we can choose to edit the program's menus, windows or documents. Let's look examine the program menus by opening the Menu Editor (see Figure 1.6).

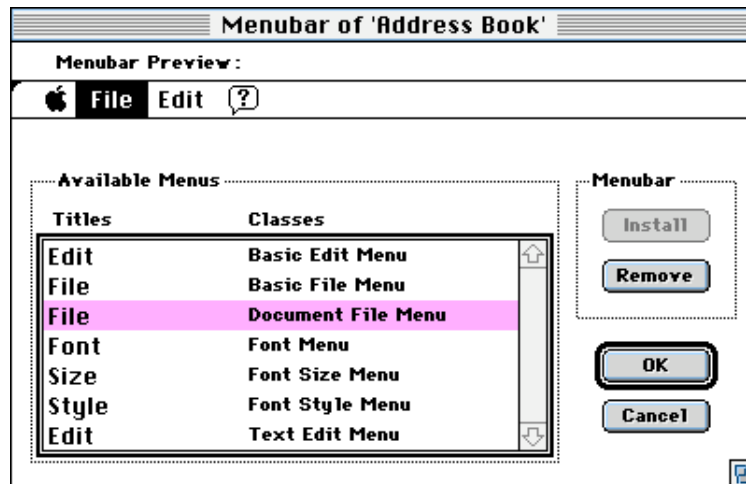


Figure 1.6: The Menubar Editor

At the top of the Menu Editor window is a preview of our program's menu bar. We've included two menus (besides the standard Apple and Help menus for Macintosh programs) -- a File menu and an Edit menu. Note that we have the choice of two varieties of pre-made File and Edit menus. We've chosen to use the Document File Menu and the Basic Edit Menu, shown in Figure 1.7.

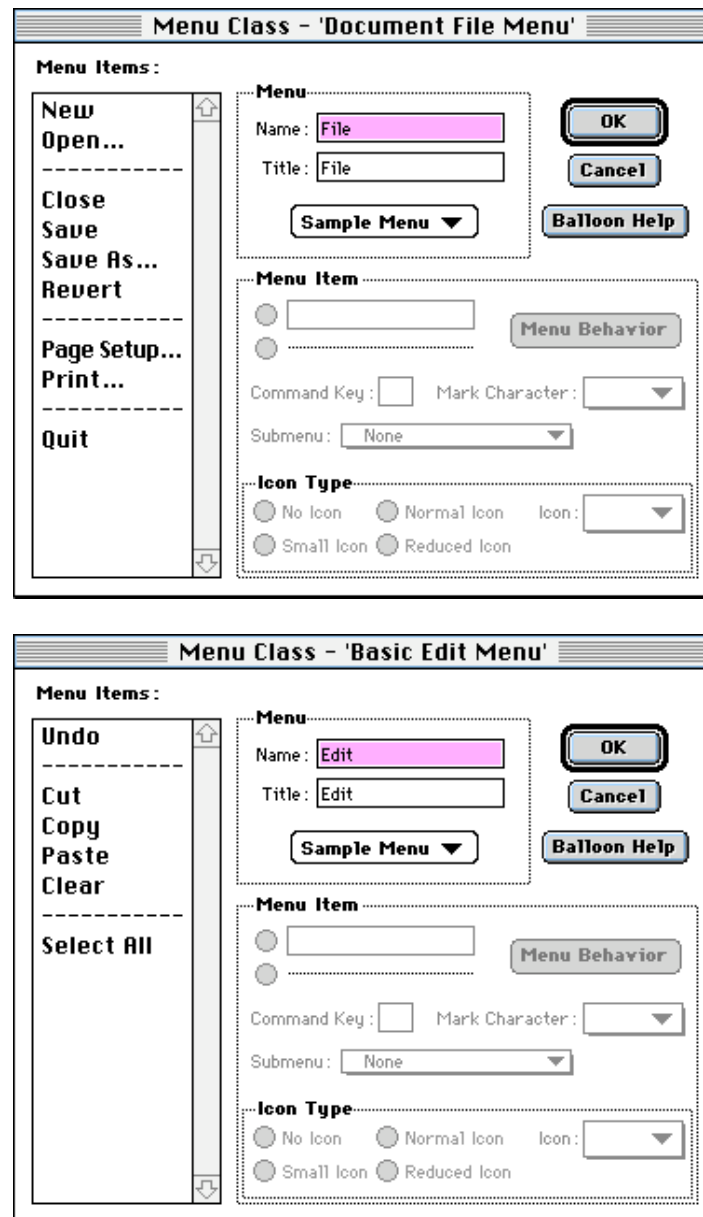


Figure 1.7: The Address Book program menus in the Menu Editor

By adding these two menus to our user interface, we've automatically added their features to our program. And what features! These two menus implement complex features such as document file saving and reading, printing and editing functions such as cut, paste and copy, to name a few. Try programming these features yourself in C, and you'll appreciate the benefits of having them done for you by Prograph.

Let's look now at the program's main window, the Get Address Window, where the individual address records of the address book will be displayed. This window is designed with the Window Editor. Window controls graphics are selected from a palette and added to the Get Address Window until it looks like Figure 1.8.

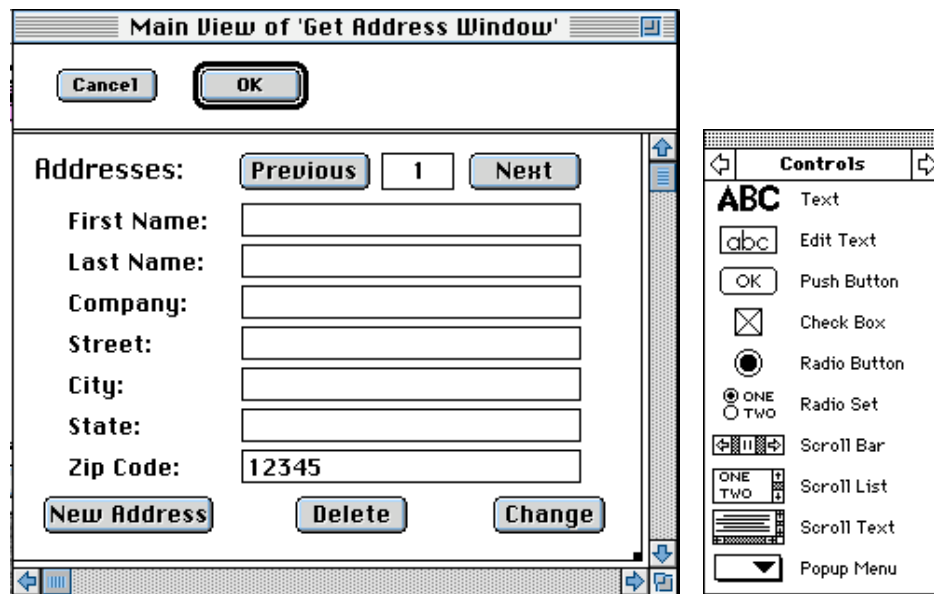


Figure 1.8: The Window Editor

The window is composed of several controls, including text editing boxes and push buttons, as well as text to label the text-editing boxes. Now the window looks the way we want it to. But we can do more than just make the window look good. We can also make each of the buttons perform a specific action or *behavior* that we desire when the program's user clicks the button with the mouse. We do this by specifying the “*Click Behavior*” of the button (see example in Figure 1.9).

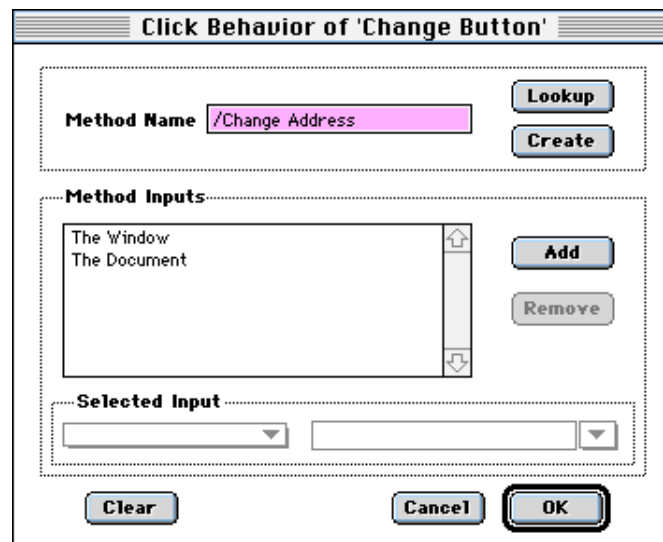


Figure 1.9: The Push Button Click Behavior Editor

In this case, we tell the push button labeled “Change” that when it’s clicked by the user, it should execute a routine called “Change Address” whose arguments are the

window containing the button and the document containing the address to be changed. We add similar behaviors to each of the other buttons in the Get Address Window -- the “Delete” button executes a routine to delete the address being displayed, “New Address” adds a new address, “Previous” displays the previous address in the address book and “Next” shows the next one. If you look back at Figure 1.7, you’ll see that the Menu editor allows you to add comparable behaviors to each item in a menu, so that when the user selects a menu item with the mouse, a particular routine is executed. So far, without writing a single line of code, we’ve created a user interface of a window and menus to our program and added the skeleton of most of its functionality by specifying the actions that the user interface will take when users select the GUI’s controls.

The last step in defining the user interface is to organize the *document* itself. You may have noticed that we haven’t defined yet exactly what a document is, although most people have a general idea of what one is. A document is any data that we want to display in a window or change, save in a file and print. Entire applications are centered around documents -- for example, a word-processing program’s document is the text you’re writing, and a graphing program’s document is the data to be plotted as well as the plot. In our case, the document in question is the set of addresses in the address book.

The document data is stored in a class called **Addresses**. Once again, classes, used in object-oriented programming, are collections of both data and the code that modifies that data. The data of the **Addresses** class will include the addresses themselves and other data needed to work with the addresses. The data members of the **Addresses** class are shown in Figure 1.10. Each address includes a contact’s first name, last name, company, street address, city, state and zip code. These aspects of an address are each given their own individual data member of the class. These data members will hold the *current* address to be displayed or printed. But what about all of the other addresses we want to remember with the address book program? They are stored in an address *list* (the first data member shown in Figure 1.10).

Lists are a *built-in* data type in the Prograph language. They are therefore available for use in any Prograph program. In other programming languages, you’d need to code lists yourself or purchase a code library that implemented lists; in Prograph, they’re thrown in for free. Each time we define a new address, we’ll just add it to the address list. When we save a file, we’ll save the entire list. We’ll work with this list shortly when we look at the code of the Address Book program.

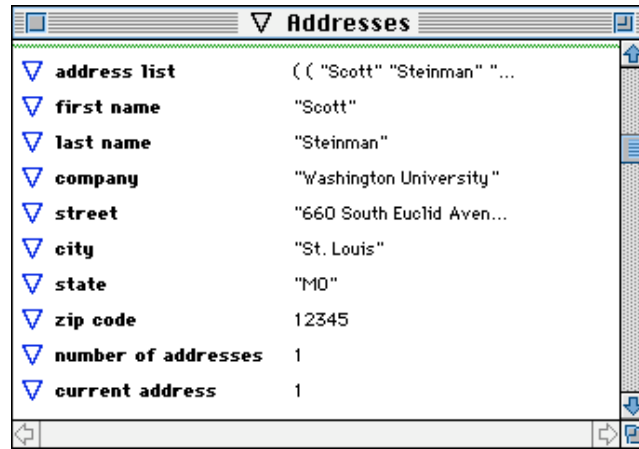


Figure 1.10: The Addresses class data

A set of classes make a document work in Prograph by managing the document's data and determining the correspondence between the document's data with the user interface, printout and file. In other words, relationships are set up between the document data and the user interface controls that display it in a window, its storage in a file and its appearance on a printed page. These relationships are called the "*mapping*" of the document data. We'll set up this mapping with the Document Editor (Figure 1.11). Notice that the editor allows us to define the data of our document -- in this case, the **Addresses** class. We then specify that this data will be displayed in the Get Address Window that we defined earlier. We also set a print layout and file type for the document data.

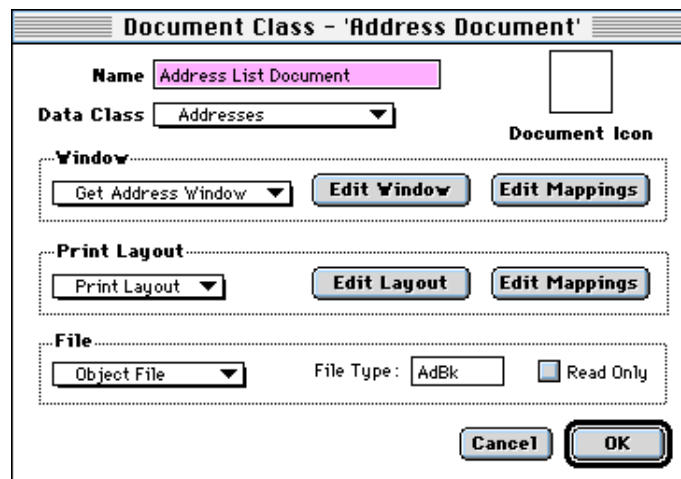


Figure 1.11: The Document Editor

Now that we've set which window will display our address data, let's state explicitly how the data will be displayed. Here we select the "Edit Mappings" button for the Window subset of the Document Editor. A Window Mapping Editor will open (Figure 1.12).

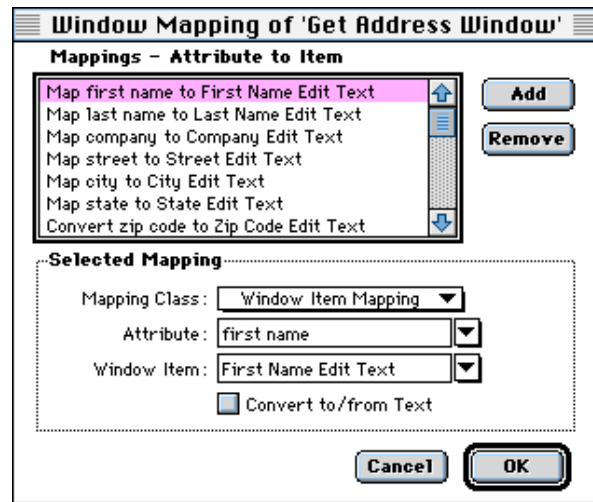


Figure 1.12: The Window Mapping Editor

In the Window Mapping editor, we set up a one-to-one correspondence between the data members of the Addresses class that hold each part of the current address and the text-editing controls of the Get Address Window that will display (and change) that data. For example, the data “first name” will be displayed in the text-editing box called “First Name Edit Text” in that window.

Isn’t this a bit tedious, setting up these correspondences? Not really -- not when you consider the alternative. In standard programming languages, displaying data in a window and changing it means writing code to transfer the data to each control of the window, managing the text in those controls, then reading that text back into the data when the window is closed or a given button is pressed. In Prograph, all we do is state which control displays which data. The rest is done for you *without your writing any code!* Any time the document’s data changes, the window display is updated for you automatically.

Hardcopy printouts are handled in a similar manner. The Print Layout Editor (Figure 1.13) lets us design the appearance of a printout of an address, including optional page headers and footers. The Page Editor, shown in Figure 1.14, is where the physical arrangement of the data on the page is planned.

Print Layout of 'Address Document'

Name Address List Layout **Page Setup**

Header

Header Type: Print View **Edit Header**

Header Height: 28 ☒ Resize Contents with Page

Page

Page Type: Print View **Edit Page**

Use Window's View: None ☒ Resize Contents with Page

Footer

Footer Type: Print View **Edit Footer**

Footer Height: 36 ☒ Resize Contents with Page

Cancel **OK**

Figure 1.13: The Print Layout Editor

Page of 'Address List Layout'

Cancel **OK**

First Name:

Last Name:

Company:

Street:

City:

State:

Zip Code: 12345

Figure 1.14: The Page Editor

Just as the address data of the document must be mapped to each control of its display window, so the data's correspondence to each field of the printout must be mapped with the Print Layout Mapping Editor (Figure 1.15). When the user selects Print from the program's File menu, the data of the current address is stuffed into each of these fields and printed out on a page.

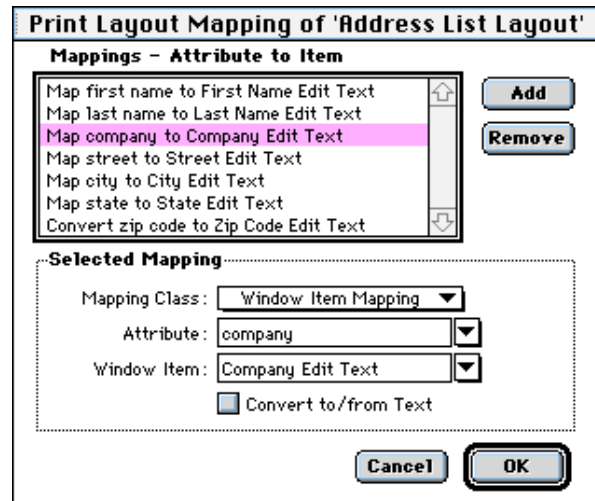


Figure 1.15: The Print Layout Mapping Editor

Each of the classes and subclasses needed for our application-specific code has been created for us by the Application Builder Editors, with the exception of the **Addresses** class. The editors have also helped us to set up the interrelationships between these classes. Without writing any code yet, we have the workings of a program that will display addresses in a window, print them out, and store them in a disk file.

Now here's the cool part! Remember all of the behaviors that we set for the push button controls of the Get Address Window earlier? Each behavior defines what code will be executed when the button is selected. This code will reside in the **Get Address Window** class to help manage that window's appearance when the code is executed. This code doesn't exist yet -- we still have to write it. But we can take a short cut -- we can let Prograph help us write the code.

If we execute the Address Book program in its present state with the Prograph code interpreter, we'll see that selecting the New item of the program's File menu will open the Get Address Window that we designed. If we then click on the Change button of that window with the mouse, Prograph will try to execute the non-existent **Change Address** routine of the **Get Address Window** class (the *behavior* we defined for that button) and we will get the following error message:

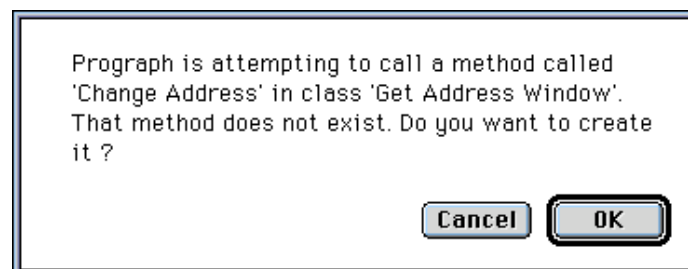


Figure 1.16: Non-existent Method error message with option to create new method

This message tells us that we're trying to execute a non-existent *method* -- Prograph's equivalent to BASIC subroutines, C functions or Pascal procedures. But the interpreter doesn't just quit or crash at this point. Prograph is much smarter and programmer-friendly than that. The interpreter gives us the option of letting it automatically access the program editor to create the method for us. Figure 1.17 shows the newly created method, already given the correct number of inputs and outputs (the dots on the bar at the top of the code window denote inputs; the lack of such dots on the bottom bar signifies that this method currently has no outputs). It's just this degree of integration between its development tools -- in this case, the editor and interpreter -- that makes Prograph so unique. What other language lets you design a user interface, run its application framework code, then help write whatever code is missing for you?

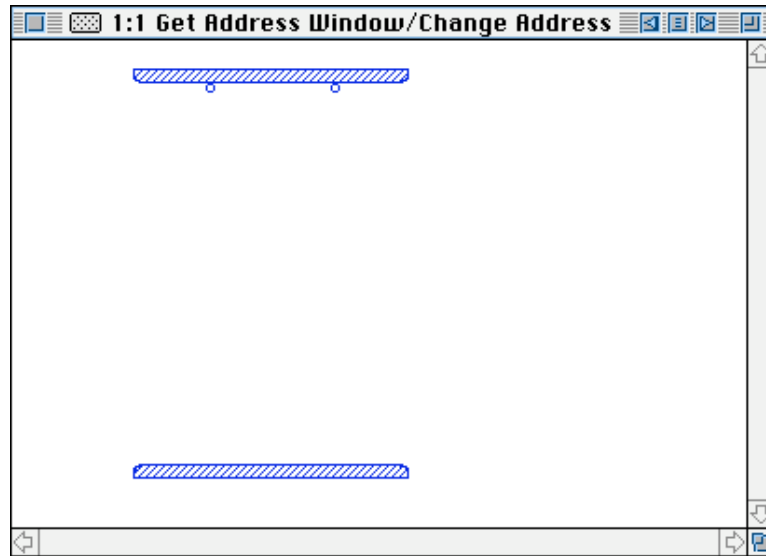


Figure 1.17: The new Change Address method of the Get Address Window class

If we repeat this process for each of the push buttons in the Get Address Window, we'll wind up with a series of methods in the Get Address Window class that will help manage that window's actions.

At this point, the program is nearly complete -- very little code has to be written. All we need to do is add the code that will perform the actions for the Get Address Window's push buttons, starting with the Change Address method, shown in Figure 1.18.

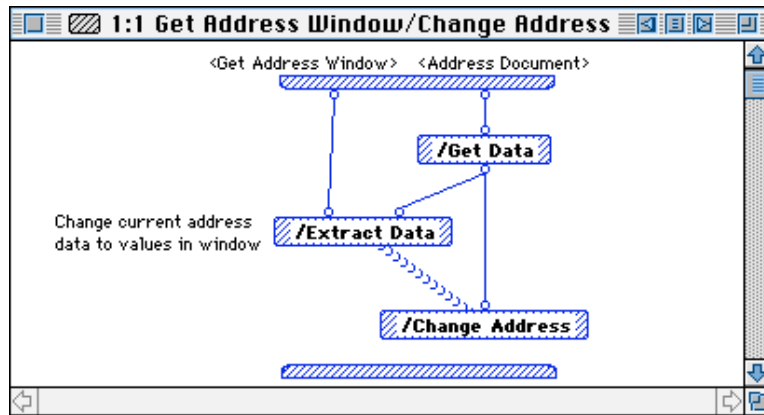


Figure 1.18: The Change Address method of the Get Address Window class

Remember that we specified two inputs to the Change Address method when we defined the *behavior* of the Change push button (see Figure 1.9) -- the window and the document involved, that is, the Get Address Window and the Address Document. This method will in turn invoke other methods to do its work. The graphical nature of the method's code makes it easy to understand. The **Addresses'** mapped data (the current address) is changed after extracting the new address from the Get Address Window's text-editing boxes. Afterwards, the displayed address is updated, as well as the current address number, by executing Display Current Address. The only methods we supply are one named Change Address in the Addresses class and Display Current Address in the Get Address Window class (the class that "owns" each method is shown by the first input into each method icon). The Get Data and Extract Data methods are supplied with the ABC framework.

How did we know what the Get Data and Extract Data methods do? The Info... menu item brings up Prograph's *on-line help facility* (see Figure 1.19), giving us information about such things as what classes our program contains, what methods and data are available in each class of the program, what the class and its methods do, what other classes interact with the class, etc. Each topic is denoted by underlined text. Each underlined item has a hypertext link to additional information.

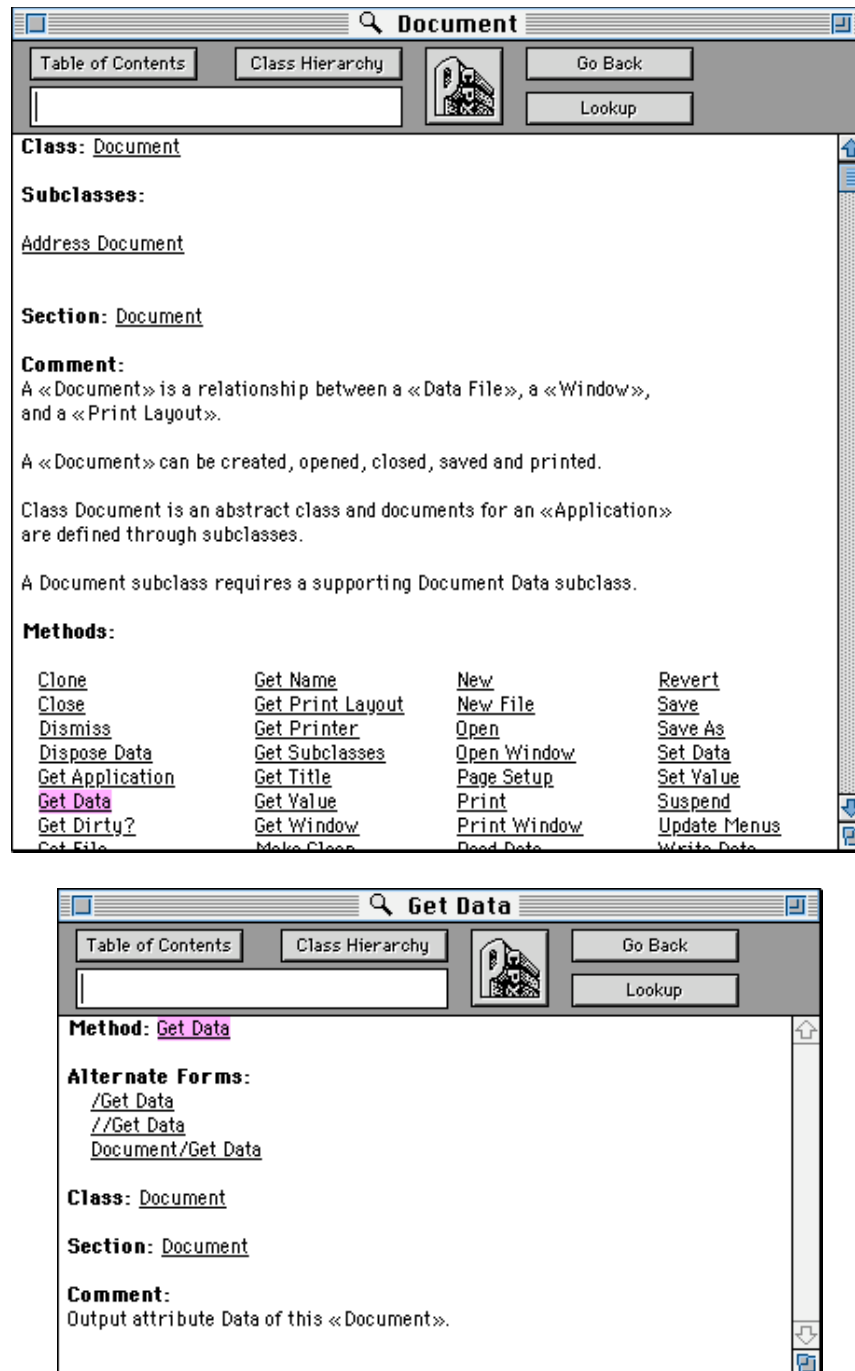


Figure 1.19: The on-line help facility listing for the Get Data method of the Document class

The Display Current Address in the Get Address Window class (Figure 1.20) updates the display of the current address in the Get Address Window. Actually, it doesn't do much at all. It just checks if the incoming data to be displayed is in a Document class or the Address Data class -- if it's the document, Get Data is called to

retrieve the data. Then the Install Data stuffs the document data back into the text-editing boxes of the Get Address Window.

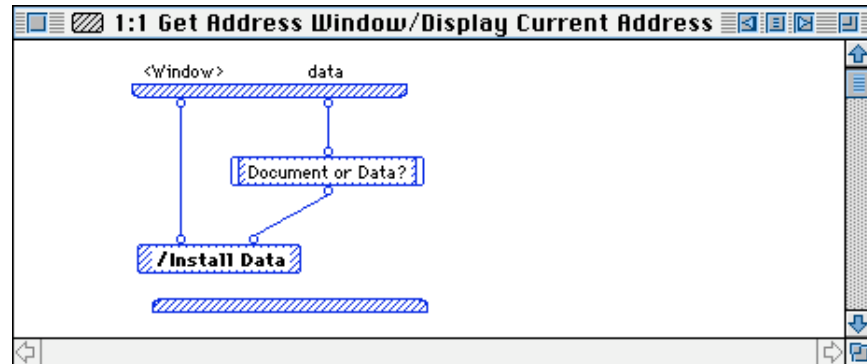


Figure 1.20: The Display Current Address method of the Get Address Window class

The icon labeled “Document or Data?” is an example of a *local* method -- a neat way of packaging and labeling code within a method that makes the method code easier to read. If we examine its code, we see that there are two alternative sets of code or *cases* that can be executed. Which one gets executed depends upon a logical test, designated by the icon consisting of a horizontal line with an “X” mark at the end of it. This logical test, called a *match*, checks what kind of data is input to the local method, as determined by the method called “type.” If the incoming data is an Address Document class, case 1 is executed, and Get Data extracts the document’s Address data. On the other hand, if the incoming data is an Addresses class, case 2 is executed, and the data of Addresses is passed through unchanged.

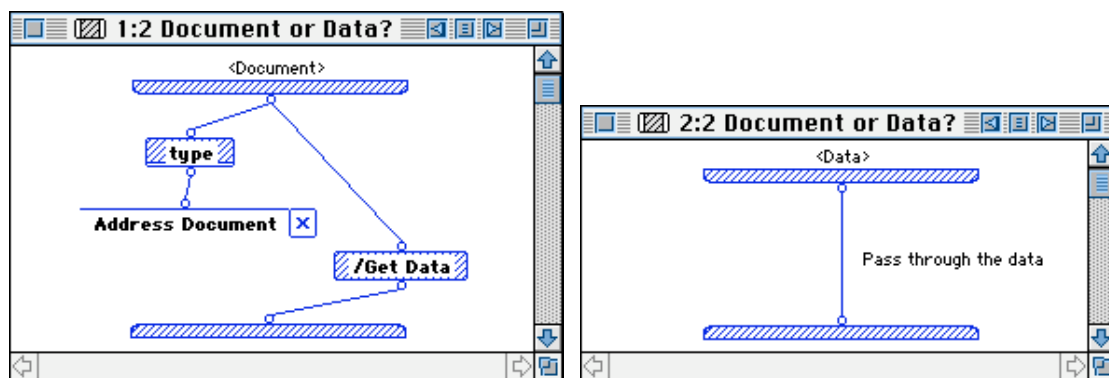


Figure 1.21: The Document or Data? local method

Look at the icon for the type method in Figure 1.21. It looks different from that of the other methods. That’s because it is. It’s a *primitive* -- one of a rich library of methods supplied with Prograph and available for your use for a wide range of tasks including mathematics, bit manipulation, string and list handling, database functions, etc. The kind folks at Prograph International have anticipated what code you’d need to use the

most and wrote it for you -- what a time-saver! The function of each primitive, what inputs and outputs each expect, and what type of data each input and output is are all available to the programmer by way of the on-line help facility.

The Figure 1.22 shows the **Change Address** of the **Addresses** class that does the actual work of changing an entry in the **address list**. It calls the **Get Current Record Contents** method, which reads the values of the data in the current address (first name, last name, company, etc.) and stuffs them into a list format using the **pack** primitive (see Figure 1.23). This list, containing the data of the current address, is made to replace one address record in the **address list** via the **set-nth!** primitive whose position in the list is defined by the **current address** number.

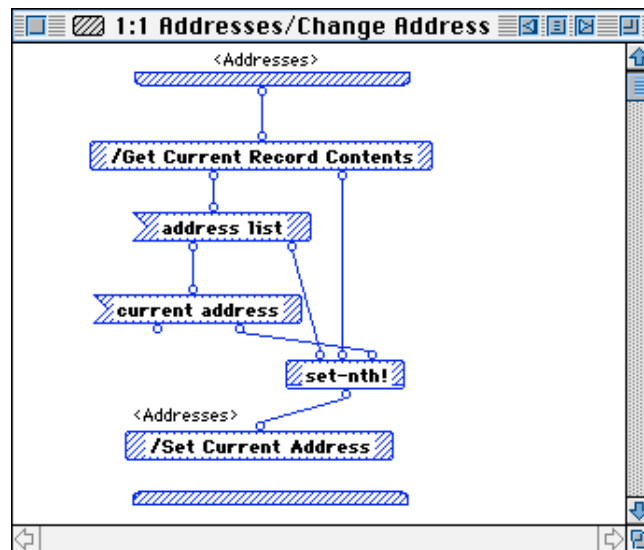


Figure 1.22: The Change Address method of the Addresses class

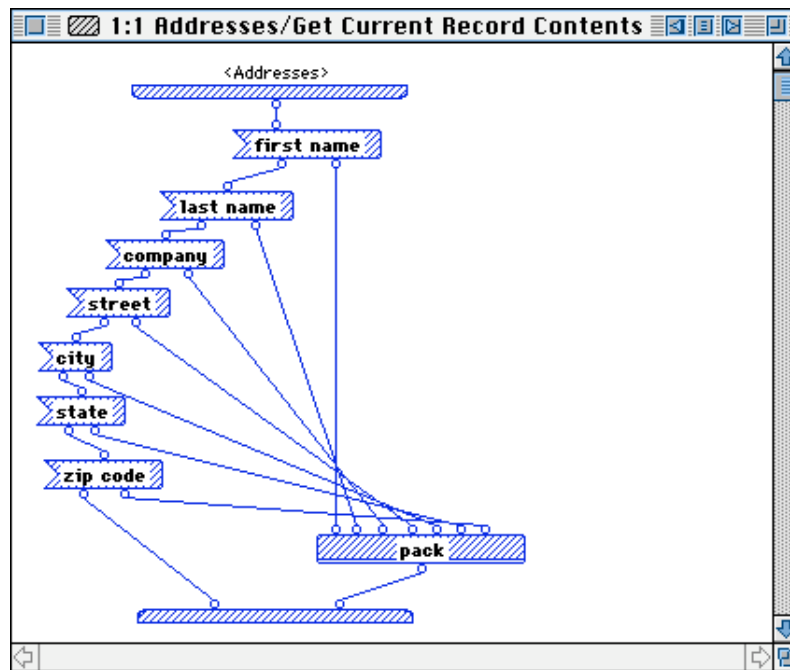


Figure 1.23: The Get Current Record Contents method of the Addresses class

The New Address method of the Get Address Window class is not very different (Figure 1.24). It extracts the current contents of the Get Address Window, adds this data to the address list, then updates the Get Address Window.

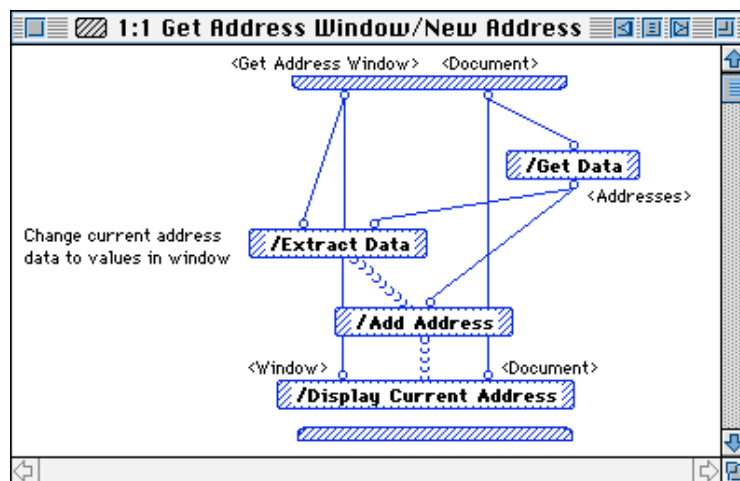


Figure 1.24: The New Address method of the Get Address Window class

What does the Add Address method do? As shown in Figure 1.25, it packs the current address data into a record list, then appends this onto the end of the address list using the `attach-r` primitive. The length of this address list is determined and the total number of addresses updated to reflect the new number of addresses in the

address book. The **current address** number is set to this new address and the current address data is refreshed for display in the Get Address Window.

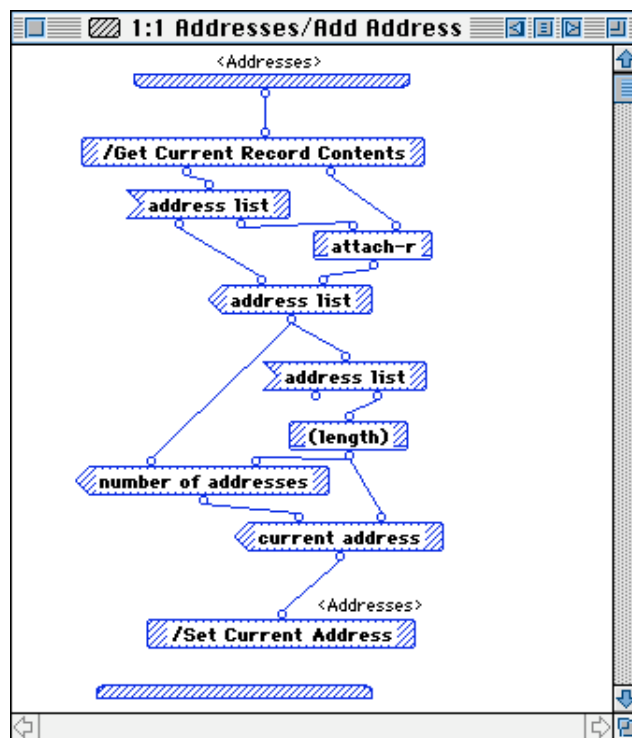


Figure 1.25: The Add Address method of the Addresses class

Delete Address is even simpler (Figure 1.26). Here we just chop out the current address record from the address list using the detach-nth primitive, then adjust the address list length and current address, and update the window display.

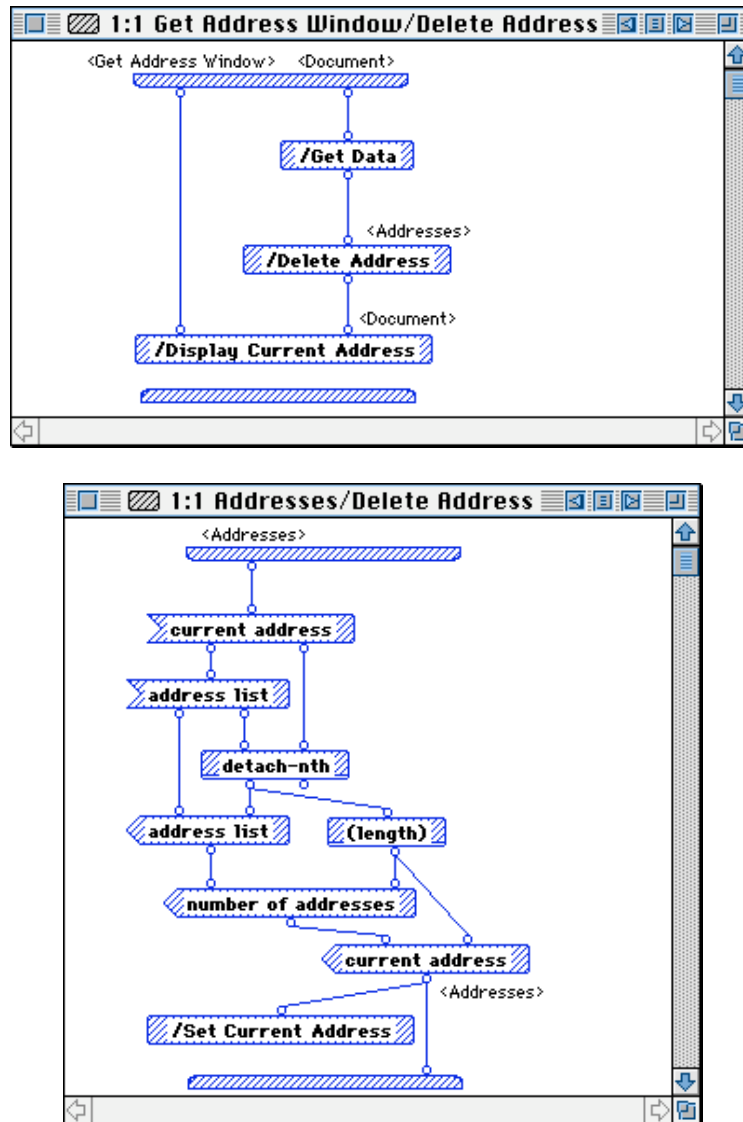


Figure 1.26: The Delete Address method of the Addresses class

Moving from address to address in the address book involves calling the **Next Address** and **Previous Address** methods of the **Delete Get Address Window** class. Since they're so similar, we'll just show the **Next Address** method (see Figure 1.27). The main purpose of this method is to call the **Addresses** class' **Next Address** method, then update the window.

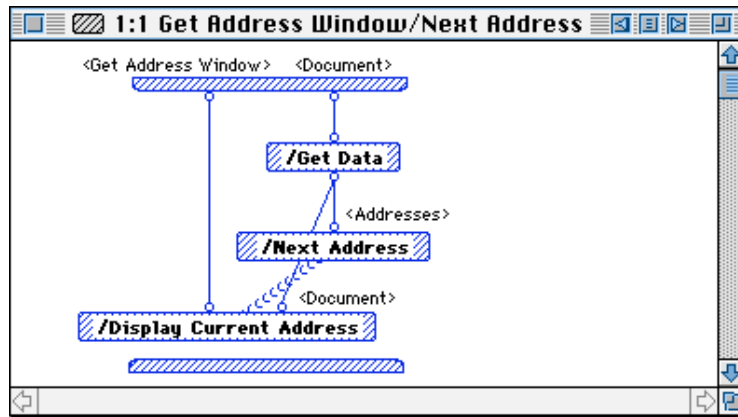


Figure 1.27: The Next Address method of the Get Address Window class

The Next Address methods of the Addresses class, shown in Figure 1.28, increments the current address number, then checks if its value is beyond the number of addresses in the address list. If it isn't, the new current address number is set and the current address data is updated. If it is, we simply stay at the end of the list.

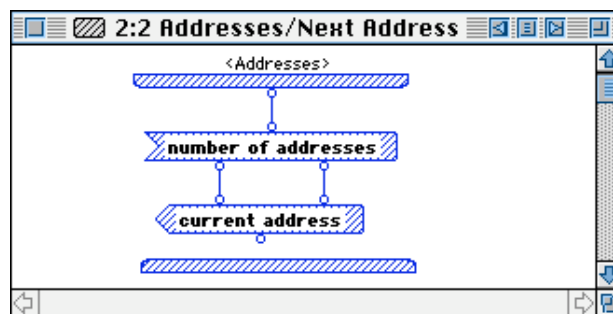
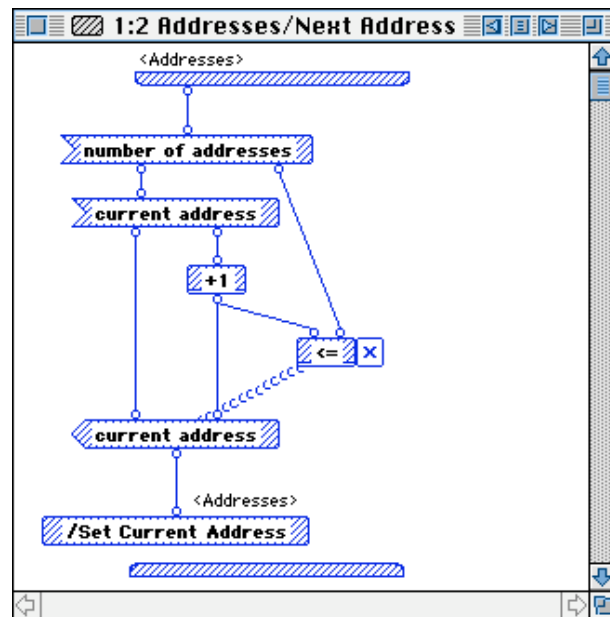
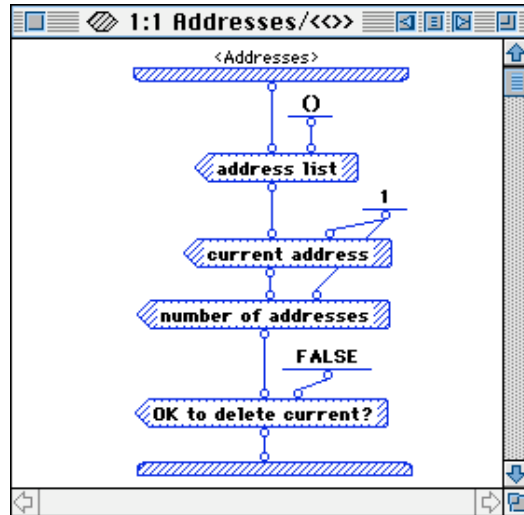


Figure 1.28: The Next Address method of the Addresses class

The last method of our program is shown in Figure 1.29. It is a special *initialization* method for the **Addresses** class. When the class is first used in the program, this method is automatically called if it exists. In this method, we set starting values for some of the data in the class. For example, we set the **address list** to be empty (the parentheses enclose the contents of the list that we set to be nothing) and the **number of addresses** to be 1.

**Figure 1.29: The Initialization method of the Addresses class**

That's all there is to the program. Not much code, is there? Yet this program does quite a lot! Think of how much C language code you'd have to write to provide the equivalent address book program. Now think about how many bugs it might have, and how many hours it would take to edit the code, compile it, run it, find a bug with a debugger, re-edit the code to fix it, recompile, try it again, etc.

Summary

This chapter has presented the advantages of the graphical Prograph language over conventional textual programming languages. The Address Book program was included in this chapter for a very important reason -- it was the *first* program written with the Application Builder Classes framework by one of the authors of this book. And yet it was written and tested in only one session at the computer. To perform similar actions in a C program requires a month or two to learn the inner workings of the computer's operating system before even starting to code the program, and once started, the program could still take days or weeks to write and debug!

The Prograph language and program development environment make all this possible. With its combination of (a) an easily understood pictorial syntax, (b) graphical display of code, data, program execution and user interface design, and (c) tight

integration and cooperation of programming tools, Prograph allows for rapid program development with a minimum of effort and debugging. In the upcoming chapters, we'll learn how to apply the strengths of Prograph for a wide range of programming tasks.