

## **14. The ABCs of Interclass Communication**

### ***Overview***

We've come a very long way since building our first example program in Prograph CPX in Chapter 1. But something's still missing. Something is very different between the Prograph programs we've described so far in this book and polished commercial software. This difference is a platform-specific *graphical user interface* (or *GUI* for short). Just as visual languages like Prograph facilitate and improve the efficiency of programmers, modern computer operating systems such as the Macintosh System 7 and PC-compatibles' Windows provide a pictorial representation of disks, directories, files, menus, windows, and controls to make the operation of the computer easier for the end user. Prograph provides all of the user interface design and handling tools you'll need to add powerful user interfaces to your programs.

Writing methods to create and manage a user interface is not necessary in Prograph as it is in languages like C or Pascal. Prograph CPX provides an excellent collection of user interface handling classes called the *Application Builder Classes*, or *ABCs* for short. In this chapter, we'll examine Prograph's visual tools for constructing user interfaces (the *Application Builder Editors* or *ABEs*) and we'll look at how the ABCs intercommunicate. In the following three chapters, we'll present examples of how the ABCs and ABEs simplify user interface construction by writing fully-functioning programs with a minimum of coding.

### ***Prograph GUI Construction Tools***

In classical procedural programming languages such as C or Pascal, programmers were forced to piece together a complex series of operating system or API routine calls to set up windows, menus, dialogs, and so forth. Every time a new program was needed, a new GUI and its supporting handler code had to be built from scratch -- a taking great deal of time and effort.

Recently, a number of object-oriented *application frameworks*, specialized class libraries for handling GUIs, have emerged. Concurrently, *code generator programs* have been made available that design user interface with pictorial tools then output source code to handle these GUIs. Even with frameworks and code generators, it is still not a simple task to design and manage a user interface with traditional programming environments. The main problem is the level of integration between these tools. Although a few C++ programming environments do contain user interface design tools such as application frameworks and code generators, they are still primarily treated as independent tools. After you design a user interface, you must still go through the edit-compile-link-debug cycle to write and test the code. If the user interface doesn't do what you want it to, you're in trouble!

A second problem is that typical application frameworks and code generators provide fairly *fixed* solutions to *flexible* problems. They treat user interfaces as just

collections of independent visual elements that are arranged on the screen until they look right and always behave in a fixed fashion. But efficient GUI programming requires much more than this. The GUI elements may intercommunicate (the theme of this chapter). However, the interdependence between the elements is often not known until the GUI is built or may change as the program is running. Static code libraries can't account for this dynamic aspect of GUI construction. In addition, the GUI handler code of frameworks or code generators must integrate seamlessly with the code you supply -- the code that makes the GUI actually carry out the actions *you* want it to. Unfortunately, these tools have always left this critical part entirely up to you. Where's the reduction of effort and programming time that these tools were supposed to give you?

The kind souls at Prograph International have designed Prograph CPX with these limitations in mind. Prograph comes with a rich set of fully-integrated yet simple to use GUI design and management tools. Included with Prograph CPX is a special object-oriented application framework called the *Application Builder Classes* (ABCs), and special GUI design tools called the *Application Builder Editors* (ABEs). These tools are tightly integrated, extremely flexible and extensible. Used together, these tools allow Prograph programmers to graphically design powerful user interfaces for their programs much quicker and without the urge to kick their dog or tear out their hair. GUI construction and programming typically takes about half the time needed with comparable tools in C++.

How is this possible? As we'll see shortly, the Application Builder Editors don't just determine how the user interface will appear. They do much more than that. The ABEs also create objects from the ABCs that will determine the behavior of new elements (like text-editing boxes, check boxes, radio button sets, push buttons, etc.) that you add to the GUI. They set the default values of the attributes of the classes that make these elements work. And, most importantly, the ABEs set the interrelationships between these classes that define how they work together as a team. Handling the GUI elements requires the addition of little new code by the programmer, which gets your program up and running in no time. More importantly, the design and coding of the user interface is fully interactive -- you can design all or part of the GUI, try it out and debug it, or add new GUI elements or new code to it even as the GUI code is executing!

### ***What are Application Frameworks?***

The key component of all GUI design and code generation tools is the *application framework* -- class libraries which encapsulate all of the typical actions of a user interface, including the presentation of menus, windows, dialogs, buttons, check boxes, radio button sets, editable text fields and popup menus. Most application frameworks also implement the main event-handling loops of applications, so that the framework itself provides most of the code needed for a full-fledged program. The programmer is only responsible for adding application-specific code, namely the specific actions taken when the user clicks the mouse or selects a menu option. Application frameworks all tend to have a core for features in common, although the precise way in which these features are implemented may differ greatly between them.

There are three ways to view application frameworks:

1. An application framework is a *class (inheritance) hierarchy* composed of a number of GUI-handling classes. In most cases, a *single* parent class is subclassed many times to form all of the user interface elements. Examples of multiple inheritance (where a class is derived from *more than one* parent class at a time) are scarce in commercial application frameworks.

2. The graphical relationship between the visible user interface elements, such as buttons or pop-up menus, makes up a second class hierarchy. This is called the *view or containment (ownership) hierarchy*. It derives its name from the arrangement of the user interface elements within each other on the screen -- one “contains” another or, stated another way, is “owned” by its enclosing element. For example, a window contains push buttons, and these buttons are in turn owned by their enveloping window. These relationships are obviously important in the visual design of the user interface.

3. Finally, the application framework must be viewed in terms of the *functional relationships* between its classes. These relationships are of two types. The first is the so-called “*chain of command*” of the user interface. When the user clicks the mouse on a user interface element (the “target” of the user action), the application framework’s code must first check to see if that GUI element’s object should handle the click or pass along that action to another object to perform an appropriate action. For example, if the user presses a key on the keyboard when a text-editing box is active in window (i.e. the current target), should that action be handled by the edit box and be interpreted as a letter of text? Or, should the keypress be ignored by the edit box and be passed along to be handled by a menu as a keyboard equivalent of a menu command? The chain of command determines which object will handle each user action.

A second functional relationship between classes is that between visible GUI elements and “behind the scenes” *non-visual* objects. An example of this is the “mapping” between how your program’s data is displayed in a window, stored in a file or printed on a page. A special non-visible GUI object called a *document* helps define the relationships between these alternative representations of your program’s data by accessing *mapping* objects, whose chore is to coordinate the actions of the individual classes that handle window displays, file access and printing.

It is the latter two major relationships within the application framework that are heavily based upon *interclass cooperation*. The application framework classes provide programs with a consistent user interface by sending messages from one user interface class to another. We stated earlier in this book that programs could be constructed as a collection of intercommunicating objects. Prograph’s own application frameworks provides a robust and powerful example of such program organization, yet an example that is easily “tamed” by the programmer.

There’s no shortage of application frameworks in the software world. In many cases, each language compiler vendor for a given machine has their own competing

framework. For example, C++ programmers of the Macintosh computer have at least four application frameworks from which to choose. Obviously, application frameworks make the programmer's task of building and maintaining a GUI easier, or they wouldn't be so plentiful. However, the class libraries that comprise most application frameworks can be quite large and difficult to grasp in terms of their inheritance relationships. Some class libraries come with a wall poster sized diagram of their inheritance tree to keep nearby as a reference when programming. This is supposed to be easy? Luckily, Prograph's Application Builder makes it easy to put a complex application framework to use quickly without having to understand all of its minute details.

### ***The ABCs as an Enhanced Application Framework***

Prograph's *Application Builder Classes (ABCs)* are the application framework used to construct user interfaces and event handlers for Prograph programs. These classes encapsulate visible user interface elements such as menus, windows, dialogs, buttons, check boxes, radio button sets, editable text fields and popup menus, as well as invisible application components like documents, data files, etc.

As with all application frameworks, the Application Builder Classes are a complex inheritance tree. However, unlike most application frameworks, Prograph's ABCs are not derived from a single base class (for example, in one such framework, all classes are derived from a class called **Object**). The Prograph CPX Application Builder Classes are a collection of 147 classes derived via single inheritance from 60 base classes. For example, all window and dialog box elements are derived from a common **Window Item** base class; all of the application's menus are subclassed from a single **Menu** abstract base class; and all files are subclassed from a **File** base class. This reduces the overhead of using a particular class -- if every class was derived from the same single base class, a lot of inherited code would be added to every single user interface class you'd use. With the ABCs, you use only the code need -- if an ABC class is not needed in the final version of your program, it is removed by the Prograph Compiler to make the compiled program smaller. The class hierarchy of the Application Builder Classes for the Macintosh computer is shown in Figure 14.1. You can also view it and navigate through it at any time by opening Prograph CPX's Info window and selecting the *Class Hierarchy* hypertext link.

Application	File	Resource
Background	Data File	B&W Pattern
Balloon Help	Object File	Color Pattern
Window Item Help	Picture File	Graphic Resource
Target Item Help	Text File	Color Icon
Toggle item Help	Style Text File	Icon
Bandor	Resource File	Finder Icon
Behavior Specifier	File Alias	Pict
Attribute Specifier	Font	Small Icon
Class Specifier	Graphic	Finder Small Icon
IAC Attribute Specifier	Line	Row
IAC Parameter Specifier	Rectangle	Screen
Menu Item Specifier	Oval	Selector

Menu Specifier	Round Rectangle	Special Keys
Method Specifier	Text	Text Editor
Persistent Specifier	Help Message	Text Filter
Window Item Specifier	Pict Res Help	Integer Filter
Value Specifier	STR Res Help	Integer List Filter
Window Specifier	String Res Help	List Filter
Clipboard	TE Res Help	Natural filter
Column	IAC Data Type	OSType Filter
Command	IAC Descriptor	Range Filter
Behavior	IAC Event	Real Filter
IAC Event Behavior	IAC Object	Utility
Menu Behavior	IAC Suite	Window
Window Item Behavior	IAC Required Suite	Window Item
Control Behavior	Marquee	Control
Task	Menu	Check Box
Deferred Task	Basic Edit Menu	Push Button
Menu Task	Basic File Menu	Radio Button
Periodic Task	Document File Menu	Scroll Bar
Text Task	Standard Apple Menu	Edit Text
Clear Task	Apple Menu	Scroll Text
Copy Task	Standard Help Menu	Graphic Item
Cut Task	Help Menu	Target Graphic Item
Paste Task	Text Edit Menu	Popup Menu
Typing Task	Text Font Menu	Print View Item
Commander	Font Menu	Date
Control Color	Font Size Menu	Page Number
Cursors	Font Style Menu	Time
Desktop	MenuBar	Scroll List
Document	Offscreen	Drag & Scroll List
Document Data	Pen	View
Basic Document Data	Print Layout	Grid
Draggor	Printer	Multiple View
Autoscroll Draggor	Rainy Day Fund	OK View
Scroll List Draggor	Regular Border	Print View
Environment	Drop Shadow	Radio Set
Event	Select Border	Selection View
Event Handler	Default Border	Autoscroll Selection View
Finder Handler	Resize	Window Item Mapping
Modal Handler	Resizer	
Multifinder Handler		

**Figure 14.1: Application Builder Class Library Hierarchy (Macintosh)**

Each Application Builder Class contains *references* to other related ABCs. These references (that is, the technique called *composition* that we discussed in Chapter 9) form the basis of the interclass communication at the core of the view hierarchy, which organizes and interrelates the user interface elements on the screen. The **Owner** attribute of several Application Builder Classes is used to refer to the next larger GUI element that holds the element in question; that is, the **Owner** of a **Button** object is the **Window** object in which the **Button** object is contained. In other words, if you need to access the **Window** in which the **Button** lies, you simply read the **Owner** attribute of the **Button**. As another example, the **Desk Top** class that represents the appearance of the monitor screen has attributes that refer to all of the program's **MenuBar** and **Windows**.

Composition is also used to implement some of the key *functional* relationships between classes. For example, the **Document** class manages text, graphical or numeric data documents, their presentation to the user and their storage. This requires some integration between the document, the windows and a printed copy on paper in which its data may be displayed, and the disk files in which it may be stored. These relationships are reflected by the presence of attributes in the **Document** class that refer to these other relevant classes -- the document class contains such attributes as **Application**, **Window**, **File**, and **Print Layout**. These attributes not only make instances of these classes readily available to code in the class methods of **Document**, but they also make these instances accessible to programmers who need to call their class methods.

Even when a related class' instance is not directly referenced, it can still be accessed by the programmer. In other words, ABCs can find functionally-related classes that do not fit any of the above categories. Other ABC classes that are not directly part of the inheritance or view hierarchy of another ABC class, and are not referenced by any of their attributes, can still be sent messages. Many ABCs contain a **Name** attribute, which is used to provide a *label* for a particular instance of each Application Builder Class. This is analogous to the newly-proposed run-time identification of the emerging C++ standard, but instead of identifying the *type* of the class, it identifies the *particular object* created from that class. This can be very useful in locating and sending a message to one given GUI element. If your program presents several windows, each window will be represented by an object created from the **Window** class, but the **Name** given to each **Window** object lets the program tell them apart.

In fact, special class methods exist for the purpose of finding a related GUI element from its **Name** attribute. For example, to locate a particular *window* with a given **Name**, all you need to do is call the **Window** class' **Find Window** method, which searches all existing windows to find the one with the name you request. To reference a particular item in a window, call the **Window**'s **Find Window Item** method giving the name of the desired window item. Similar methods exist for locating a desired menu or menu item. Likewise, to send a message from one window element to another (such as changing the contents of a Text object depending upon the current selection of a radio button set), call the first object's **Find Window** method to access its containing window, then find the second element with **Find Window Item**. These class methods help to set up more indirect messaging between objects. Let's say you want to enable a menu item when a check box is selected in a dialog box (a special type of window). A menu is not in the inheritance hierarchy or the view hierarchy of a dialog box, nor is it referenced by an attribute in the **Window** class. But we can still access the menu item. First, the long method. When the check box is clicked, we can find the enclosing view of the check box (the dialog box window) from the **Check Box** object's **Owner** attribute. The **Window**'s **Owner** is the **Application**. From the **Application**, we get to the **Desk Top**, and then to the **MenuBar**, all via **Owner** attributes. Once we have the reference to a **MenuBar** object, we can call its **Find Menu Item** method to access the particular menu item we want to enable. While this seems a little roundabout, remember that without these

references, there would be no way at all to access the menu item from within a dialog box. But before you get frustrated, here's the short method. Simply get the value of the **The Application** persistent. This lets you skip stepping through the view hierarchy of the check box and go directly to accessing the menu item.

Finally, there is another very powerful method of setting up interobject communication, both among the objects within the ABCs and between the ABCs and your own application-specific objects. This mechanism is known as *behaviors*, a form of mapping between an ABC object and the application-specific code it will execute. Since the application framework cannot always know in advance which of your methods will be called or what inputs your methods will require, it uses a special mechanism that involves *inject* constructs, first discussed in Chapter 4. Remember that injects allow you to determine the name of a method that you want to call at *run-time*, rather than when building the program. This lets the ABCs call one of your methods without needing to find out where in the ABC code your method would be called and hard-coding the name of your method into that ABC code. Such flexibility would not be possible without injects. We'll return to this when we discuss behaviors.

We'll be looking briefly at the workings of the most often called Prograph Application Builder Class methods and most-accessed attributes to see how interclass communication helps to manage complex user interfaces. Afterwards, we'll take a look at how the Application Builder Editors themselves setting up interclass relationships transparently for the programmer.

### ***The Application Builder Classes and Interclass Communication***

The Application Builder Classes provide a high-level system of interacting objects that interact with the low-level operating system routine calls that manage the user interface. By using the ABCs, the programmer rarely has to call operating system routines directly, since they are encapsulated into the ABCs. In addition, an ABC Starter Project is provided to serve as a fairly complete "skeleton" application. To create your own application, all you need do is add new application-specific windows, menus and documents to the starter project. The result is faster application prototyping and development, as well as easier program maintenance.

The ABC library is centered about a class called **Application** that coordinates the execution of the program. The entry point for a Prograph program based upon the ABCs is a universal method that is itself called **Application**, in the ABC's Application section. This method receives as input an instance of the **Application** class stored in the **TheApplication** persistent. This object is then sent an **Initial** message, which gets the application running by allocating memory, creating objects, getting program resources, and opening initial windows. The main event loop is then entered by executing the **Run** class method. This loop is executed until the user chooses to quit the program. At that point, the **Close** class method is executed to perform clean-up operations such as deallocating memory and closing windows, then the program ends.

Although a single **Application** class is responsible for getting the program up and running, application execution involves many more classes than this. The **Application** class must create instances of several other helper classes, such as **Desktop**, **Commander**, **Menu Task**, **Printer** and **Clipboard**, then send messages to these classes to request actions that will handle the user interface and other aspects of the program. The **Application** class has been designed to do just that. Many of its attributes are references to these other classes (you can find these easily since these attributes tend to have the same names as the classes to which they refer). These references are critical for the operation of the program. For example, the **Desktop** attribute is used to send an **Open** message to the **Desktop** class which, among its other tasks, opens the program's initial windows by referring back to the **Application**. Without these references, an excess of persistents would have to be amassed to hold instances of each class. It would require too much space to describe *every* example of interclass communication within the ABCs. We leave the exploration of the multitude of attributes and methods of the ABCs to the readers if they are so motivated (a detailed description of the ABCs is provided in the Prograph CPX ABC reference manual as well as in the Info on-line help system). Instead, let's look now at how the programmer can build interclass communication with Prograph's built-in user interface editors.

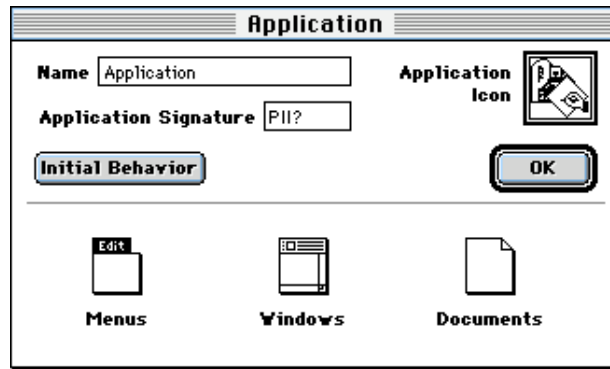
### ***The Application Builder Editors as GUI Designers***

The Application Builder Editors (ABEs) are a series of graphical user interface design tools. An editor exists for each major GUI element encapsulated in the ABCs. For example, there are Menu Editors for menus, Scrolling List Editors for scrolling lists, etc. With these editors, a programmer can design the look and feel of an application's GUI with graphical tools. The tools provide support for building user interfaces at several levels. At their simplest level, the ABEs enable the application's menus, windows, dialog boxes and documents to be designed and rearranged until they look just right and present information to the user in the most intuitive manner.

The ABCs are also extendible. If you write a subclass of one of the ABC classes to add new GUI functionality, you may add your new class to those already available to the ABEs. This lets you not only add your new GUI element to future programs, but it also allows you to edit the attributes of the new class visually as you design user interfaces with the ABEs. We will show you how to add programmer-defined GUI elements in the following chapters.

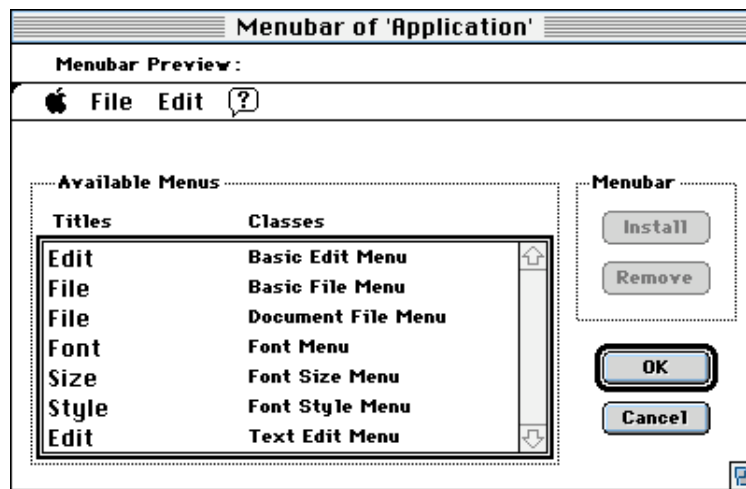
The ABEs are entered by selecting the Edit Application... menu item. The first editor you'll encounter is the *Application* editor, shown in Figure 14.2. This editor is the starting point for GUI design. It's from here that you'll enter the other editors for menus, windows and documents, as well as provide a name and program icon for your application.





**Figure 14.2: The Application editor**

The *Menubar* editor (Figure 14.3) allows you to add individual menus to an application's menu bar. By default, the application's menu bar will be given a File menu and an Edit menu. If your program does not need to perform any more actions than the usual application menus provide, you can easily just use one of the pre-built menus listed in the scrolling list of menus of the Menubar editor. Otherwise, create your own menu from scratch or subclass one of the existing menu classes to build a more specialized but related menu.



**Figure 14.3: The Menubar editor**

Individual menus are created and modified with the *Menu* editor, seen in Figure 14.4. The specific menu items are added to each menu in the menu bar, along with an icon or keyboard command equivalent for the menu item, if desired. Submenus may be added to any menu item to form hierarchical menus.

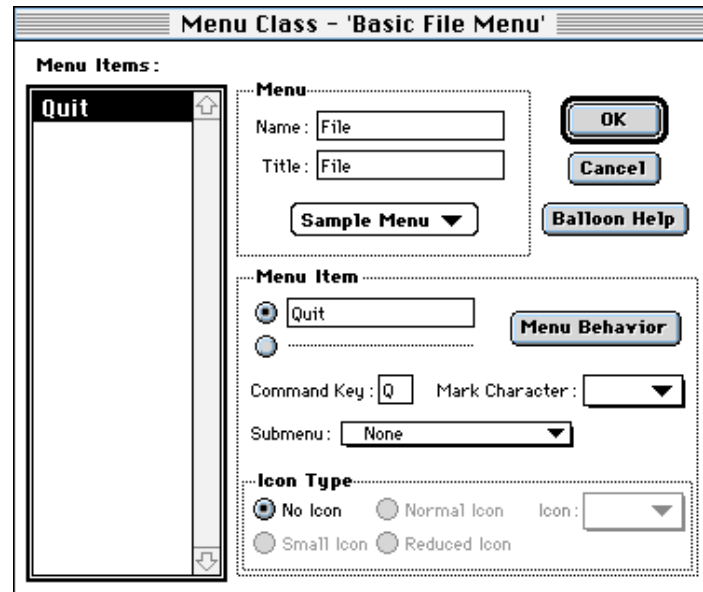


Figure 14.4: The Menu editor

Windows and dialog boxes are a little bit more involved than menus, but the *Window* editor (Figure 14.5) and its related editors help keep them relatively simple. At the topmost level of these editors is the *Windows* editor, which lists all available windows for your program. When a particular window title is clicked in this list, its size and position on the monitor screen are depicted in the editor window. Selecting that window opens up its *View* editor.

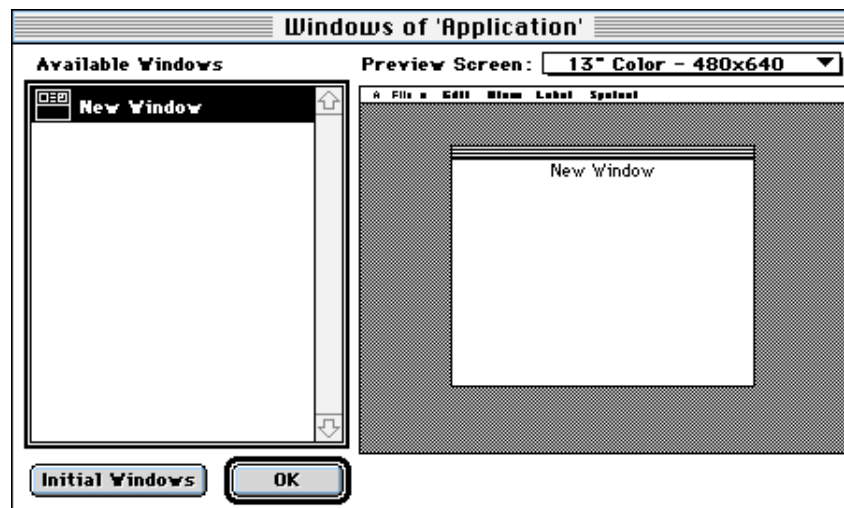
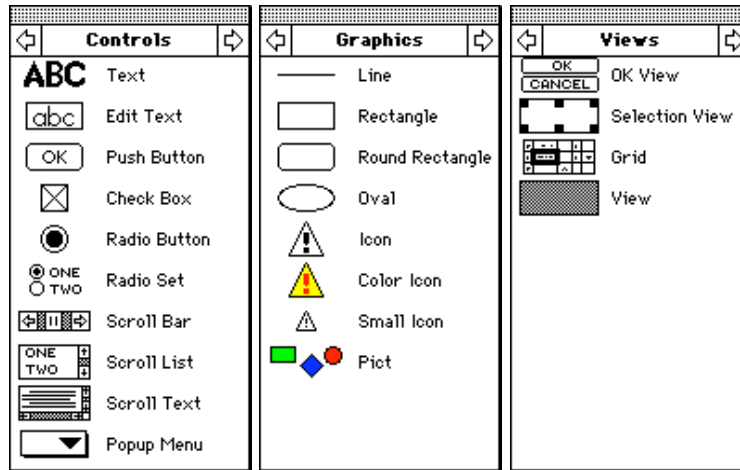


Figure 14.5: The Window editor

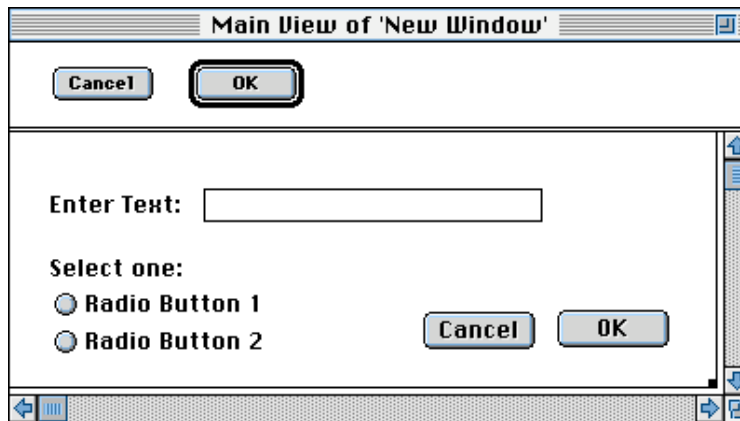
When the *View* editor for an individual window is opened, a floating palette appears that contains window items such as controls and graphics that may be displayed in the window. Figure 14.6 shows the tools available on this palette. You can create your own controls, graphics or views by subclassing the ABC classes, then add it to these

floating editor palettes to easily add it to future programs. In the upcoming chapters, we'll create our own special Push Buttons and Views and add them to these palettes.



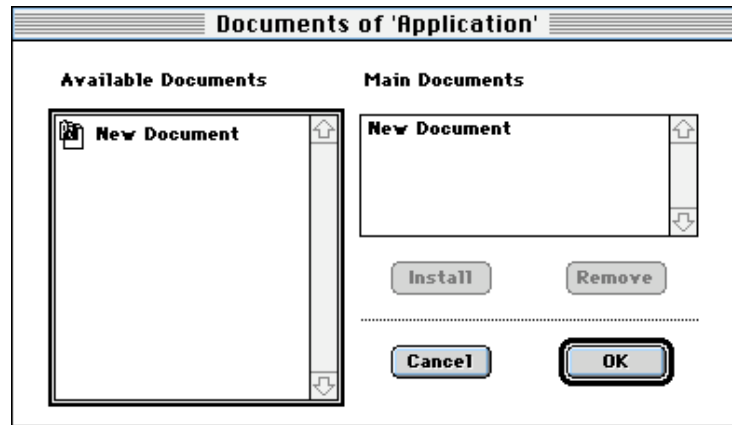
**Figure 14.6: The Window editor palettes**

The window we are designing is displayed graphically in the *View* editor, shown in Figure 14.7. The controls, graphics and views of the floating palette are placed in the window by dragging their icons from the palette to the *View* editor's representation of the window.



**Figure 14.7: The View editor**

The final major editor is the *Document* editor (Figure 14.8), which is used to define new documents. We'll return to this editor shortly when we discuss documents.

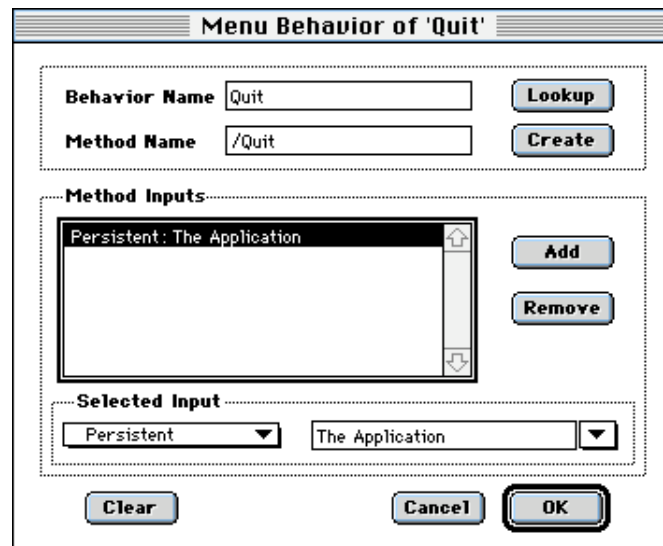


**Figure 14.8: The Document editor**

### *The Application Builder Editors as Action Designers*

To the programmer, the Application Builder Editors behave very much like any other GUI design tool -- you drag icons of the GUI elements onto the screen and arrange them until they look nice. As useful as the ABEs are as visual tools for graphical user interface design, this is just the tip of the iceberg. The Application Builder Editors don't just arrange GUI elements in a window or menu items in a menu -- they *also help set up requests for action (method calls) to and between the classes that make the user interface* so the GUI will perform as it should. That is, the ABEs define the view hierarchy and functional relationships between GUI objects as you design your application-specific user interface.

Let's start with the *Menu* editor (refer to Figure 14.4). Note the button labeled "Menu Behavior". What's a *behavior*? Behaviors are simply the *actions* taken when a control or a menu is selected by the user. In other words, behaviors are the *requests for action* passed to the GUI element object or to another functionally-related object. We are requesting that the object perform the action by calling one of its class methods. We set up this communication between the classes by using the graphical *Behavior* editor, shown in Figure 14.9.



**Figure 14.9: The Menu Behavior editor**

The ABE's behavior editors give us a lot of flexibility in writing our application-specific code. We don't have to tailor our own methods to fit some rigid requirements of an application framework, such as only being able to write methods with fixed names or a fixed number and type of inputs to interact with the framework's code. Behaviors instead make the framework adapt to *our* code! How is this possible?

In each Behavior Editor, we enter a name for the behavior and define and name a class method (either written by us or within the ABCs already) that will be called to accomplish the behavior. This specification of the method to be called when a user interface element is selected ensures tight integration between the Application Builder Classes framework and our own code. In fact, by clicking the *Create* button now, you could write the code for this new class method without exiting the editor, another example of how Prograph lets you create programs faster by integrating all of its program design tools. When we define the behavior, the Behavior Editor automatically creates an instance of a subclass of the class **Behavior** for us and places a *reference* to it in the GUI element's object. There is a subclass of **Behavior** for each major user-selectable GUI element, including **Menu Behavior**, **Window Item Behavior** and **Control Behavior**. The **Behavior** classes therefore form the "glue" between the GUI element and the method we want it to call.

More than just naming the method to be called, the editor also allows us to set the *number and type of inputs to this method*. The **Behavior** classes do their work in concert with a set of classes derived from **Behavior Specifier**. The subclasses of **Behavior Specifier** include **Window Specifier**, **Window Item Specifier**, **Menu Specifier**, **Menu Item Specifier**, **Class Specifier**, **Method Specifier**, **Attribute Specifier**, **Persistent Specifier** and **Value Specifier**. Each subclass corresponds to a possible input data type for the method called as a behavior. Popup menus in the editor present us with a choice of the type of input, as well as some specific instances of these input type. Objects of the

subclasses of **Behavior Specifier** are also created for us along with the **Behavior** class by the Behavior Editor when we define a behavior. The **Behavior** object contains as one of its attributes a list of these **Behavior Specifier** objects, one for each input to the method.

So between the Behavior Editor and the **Behavior** and **Behavior Specifier** objects it creates for us, we've defined both what method will be called when a GUI element is selected by the user and what inputs the method expects. What if we decide to change the method or its inputs? No problem -- the Behavior editor will take care of it for us. Most application frameworks couldn't possibly do this. We'd have to change our code ourselves to account for even the slightest change in behavior of a GUI element.

How does the application actually handle all of this at run-time? When a GUI element is selected, its referenced **Behavior** object's **Do** method is called (see Figure 14.10). In the **Do** method, the name of the method to be called is retrieved from the **Behavior** object's **Method** attribute, and a list of **Behavior Specifier** objects is retrieved from the **Behavior** object's **Specifiers** attribute via the **Resolve** method. **Resolve** is a class method of **Behavior Specifier** (Figure 14.11) that finds the proper input objects, variables or constants referenced by each **Behavior Specifier** object by sending the **value** attribute of the **Behavior** object into an inject. Finally, the method named by the **Behavior** is called by the **call** primitive, which receives as input both the method name and a list of the method inputs. The **call** primitive also acts somewhat like an inject construct, in that doesn't know the name of the method it will call until run-time.

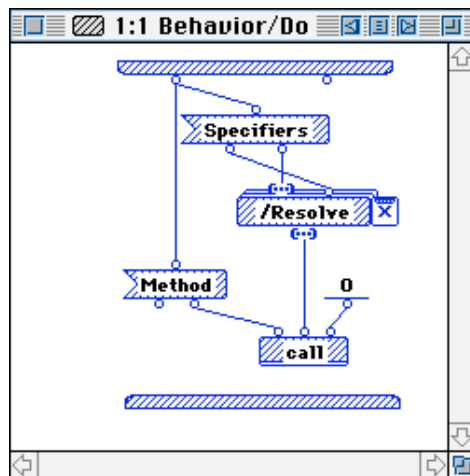
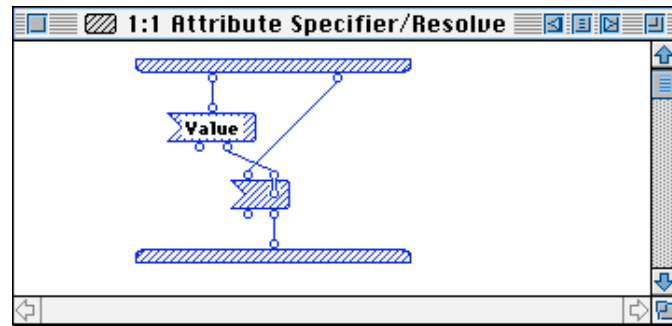


Figure 14.10: The Do method of the Behavior class

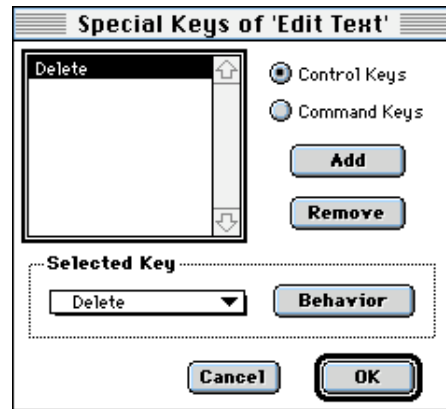


**Figure 14.11: The Resolve method of a Specifier subclass**

As an example, Figure 14.9 shows that when the Quit menu item in the File menu is selected, a class method named **Quit** will be called as its behavior. This method expects one input -- a persistent called "TheApplication", which contains an instance of the **Application** class (as shown by the selected input of the **Quit** method in that figure). In other words, this behavior sends a method call request through which the **Menu** class communicates with the **Application** class. The ABEs have let us set up interclass communication easily with a few clicks of the mouse.

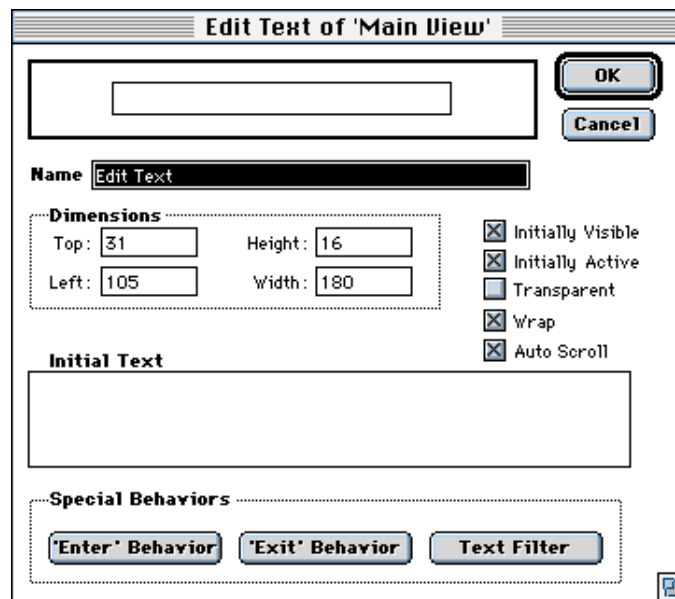
Controls in windows, unlike menu items, have different behaviors that are specific to their type of user interface element. Many controls have actions only when they are clicked on with the mouse. These controls therefore have only one behavior -- a *click behavior*. Others perform actions either when they are clicked on or when their graphics must be refreshed. For example, a grid display (a 2-dimensional grid of text or icons) within a window might be redrawn if the window is inactivated then reactivated. Controls such as these have a *draw behavior* in addition to their click behaviors. The Behavior Editors for these two types of behaviors are accessed by selecting the Click Behavior... or Draw Behavior... menu items.

Other user interface elements must take advantage of additional behaviors. For example, text editing boxes might need to detect when the user presses a particular key and perform a special action. The Special Keys menu item allows the programmer to define which keys will be handled in this manner and define a behavior to do so for each of these keys (see Figure 14.12).



**Figure 14.12: The Special Keys editor**

Data being displayed in a text editing box may be changed in its format before being displayed or after modification by the user. Changes in the value of the data might also affect other data in the program. To account for these possibilities, text editing boxes are given two important behaviors -- the Enter behavior and the Exit behavior (see Figure 14.13). As their names imply, they allow for actions to be taken (messages to be passed) when the text editing box is first selected and when it is exited because the user clicks in some other user interface element. For example, the Exit behavior might call a method that will confirm that the text you've entered in the text editing box has the proper value for your application before it lets you exit the text editing box.

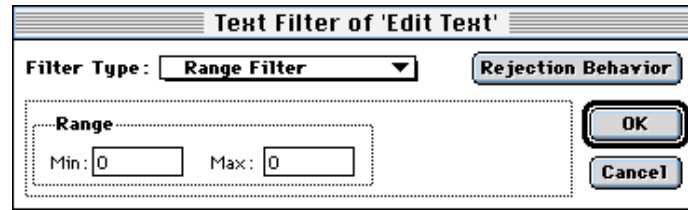


**Figure 14.13: The Edit Text Behavior editor**

Note the third button at the bottom of the *Edit Text* editor in Figure 14.13. This button brings up a *Text Filter* editor, shown in Figure 14.14. Text filters are methods that are called to enforce that a specific type of data be entered by the user into the text editing



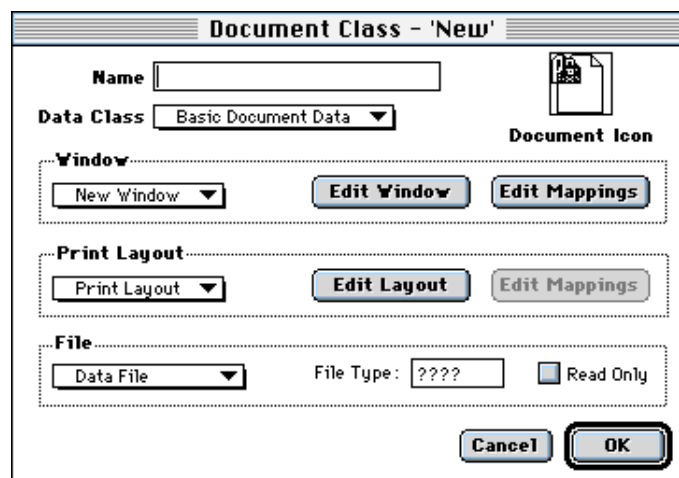
box. Associated with text filters are a *Rejection* behavior -- the action to be taken when an incorrect data type or data outside of a given range is entered by the user.



**Figure 14.14: The Text Filter editor**

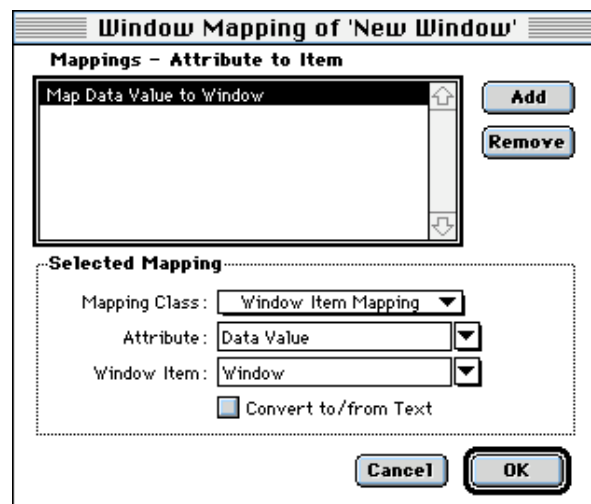
Documents are just as easy to design as menus, windows and dialog boxes. The Document editor (Figure 14.15) helps define the interrelationships between the document, the window in which its data is displayed, the print layout for its hardcopy representation and the storage format for writing its data to a file. *Mappings* play an important role in these interrelationships.

The Document Editor contains several popup menus that help you define the classes that will be used to help the newly defined Document class perform its duties. For example, the popup menu entitled Data Class only lets you select subclasses of Document Data to serve as the class in which the document's data is stored. The Window section of the editor contains a popup menu that lists all available windows in the program. The Print Layout section of the editor contains a popup menu that only lets you select the Print Layout class or subclasses of it that you might create. Finally, the File section determines the data file format in which to store the document's data -- data file, object file, picture file, text file or styled text file. The data file requires you to write code to store the data item by item. Each of the other file types would be stored by the Document class for you automatically. They store the document's data as text, styled text (text with formatting and styles such as italics or boldface), or pictures. The object file type is especially useful since it just saves (and later reads back) all of the attributes of the Document Data subclass as a whole.

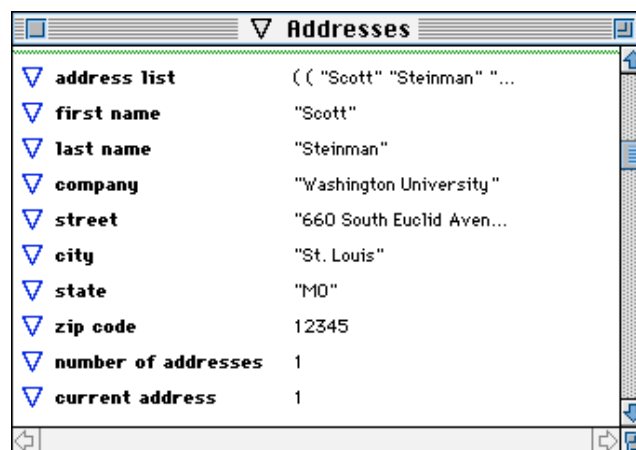


**Figure 14.15: The Document editor**

Mapping is simply a one-to-one link between each data item (a single variable or list) and the window element or print layout location that displays it. The *Window Mapping* editor is shown in Figure 14.16. In this editor, document data is associated with the particular window item that will present it to the user by selecting via pop-up menus the data attributes and the window items to display them. In addition, the programmer may select a class to implement the mapping between a document and its display window. The *Window Item Mapping* class is a special helper class that is provided for us in the ABCs, but we may also choose to use a subclass that we define to do the mapping instead.

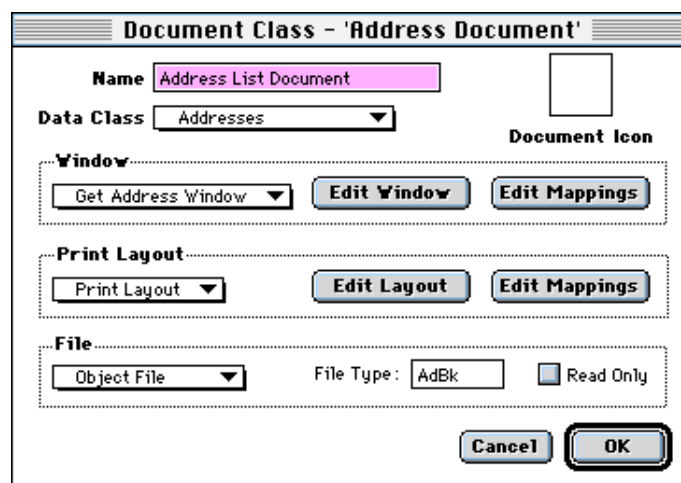
**Figure 14.16: The Mapping editor**

As a concrete example, let's return to the definition and mapping of the document used in the Address Book example program of the first chapter. The document's data was stored as attributes of a class named **Addresses**, shown in Figure 14.17.

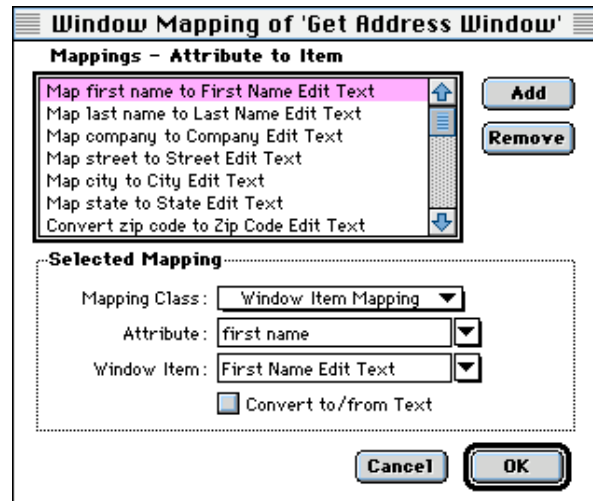


**Figure 14.17: The Addresses class data**

The document for the program was named Address List Document, and was implemented by the class of the same name. In Figure 14.18, the Document Editor is shown with the settings for the Address List Document, whose name is typed into the Name text editing box of the editor. At the top of the editor, we also see this document uses the **Addresses** class as its data class. In the Window section of the editor, we've set the **Get Address Window** as the window that will present the address data to the user. The **Print Layout** class of the ABCs will provide printing capabilities to the document, and by selecting the **Object File** setting in the File section of the editor, we have specified that we will save all of the contents of the **Address** object (all of its attributes) in files created by this application.

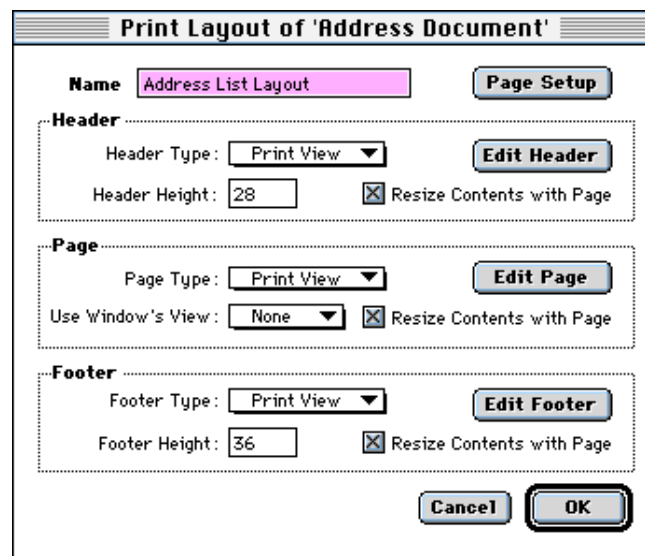
**Figure 14.18: Definition of the Address Document in the Document Editor**

Although we've set which window will display our address data with the Document Editor, we must also explicitly specify *how* the data will be displayed in that window. We select the "Edit Mappings" button for the Window subset of the Document Editor to enter the Window Mapping Editor (Figure 14.19). Here we associate each attribute of **Address** to the Window Item of the Get Address Window that will display the attribute's data. The Add button of the editor creates a new mapping. We then highlight the new mapping in the scrolling list of mappings at the top of the editor and define the mapping in the Selected Mapping section of the editor. First we specify the class that will perform these mappings. By default, only the **Window Item Mapping** class can be selected. Next we select the name of the **Address** attribute to be mapped from the Attribute pop-up menu, which lists all of the attributes of that class. Finally, we choose the window item that will display the data with the Window Item pop-up menu, which lists all of the user interface elements in the Get Address Window.



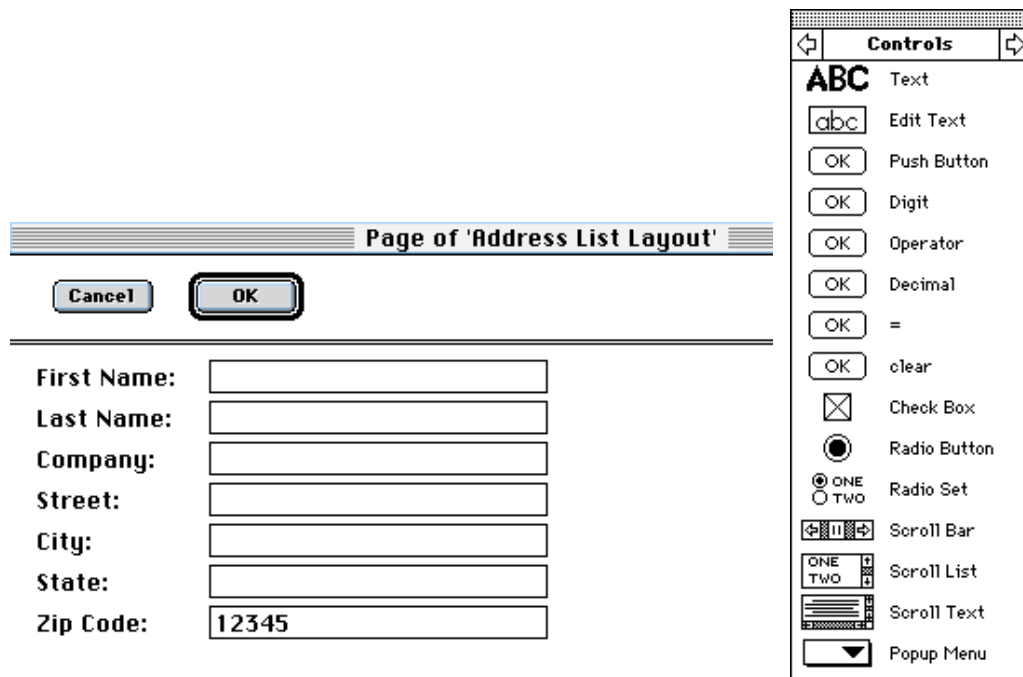
**Figure 14.19: Definition of the mapping of the Address Document data into the Get Address Window with the Window Mapping Editor**

Printouts of the document data are defined with the Print Layout Editor (Figure 14.20), which lets us specify the appearance of a printout of a single address. We can also add a header or footer to the page. Headers and footers may contain text or graphics of our own design, or the current date or a page number. These items may be added to the header or footer using palettes similar to those of the View Editor. Instead of showing you these editors, we'll move ahead to the Page Editor, which lays out the appearance of the main body of the printed page.



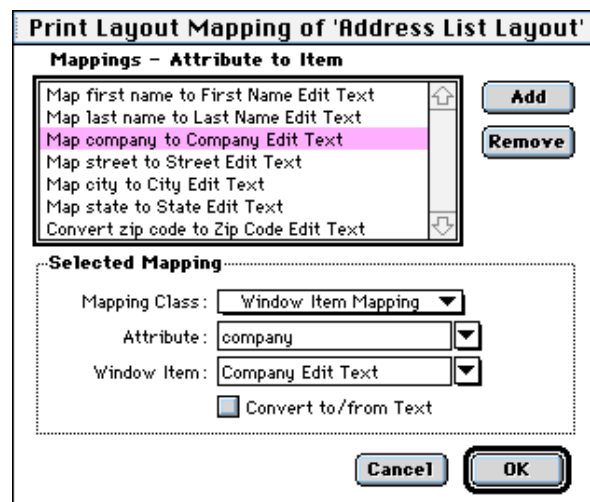
**Figure 14.20: Definition of the printout of the Address Document data with the Print Layout Editor**

The Page Editor, shown in Figure 14.21, is where the physical arrangement of the data on the page is planned. We just draw graphical elements onto the page layout from a palette just as we did to define a window.



**Figure 14.21: Arranging the layout of the Address Document printout with the Page Editor**

Just as the address data of the document must be mapped to each control of its display window, we must also map the data's correspondence to each field of the printout with the Print Layout Mapping Editor (Figure 14.22). 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 14.22: Mapping the Address Document data with fields of the printout with the Print Layout Mapping Editor

Before we finish our discussion of behavior editors, we should mention one special behavior called the *Initial* behavior. This behavior, set in the *Application* editor (see Figure 14.2), is an action taken when any application first starts up, even before any windows are opened. Initial behaviors lets us do things like initialize variables and set default values from those stored in a file, or set up interprogram communication.

The Application Builder Editors are an advanced means of managing a complex class library -- powerful tools for defining interclass messaging with a minimum of effort. What's probably more important to the programmer is that after using the ABEs, the programmer has to write very little program code to fully integrate the user interface into an application. All that essentially remains is getting the data to be presented and manipulated into and out of the user interface. As an example, if a dialog box is to be presented to change program settings in an application without documents, the only user interface code that the programmer has to write is code to put the values of the settings into the GUI elements (text editing boxes, check boxes, radio buttons, popup menus, etc.) when the dialog box is first presented, then write the code to retrieve these values when the dialog box is dismissed. If the dialog box were part of a document-based program and were displaying data from the document, even this minimal amount of code-writing would be unnecessary, since the *Document* and *Mapping* classes themselves would handle setting and getting the dialog box's values.

### Summary

In this chapter, we've reviewed the underlying workings of the Application Builder Classes and their associated Application Builder Editors. These tools set up the complex interclass messaging protocols necessary for managing powerful and flexible user interfaces. What this means for you -- the programmer -- is that less work is required on your part to design and manage a professional-looking and commercial-quality GUI.

- The ABCs are a special *application framework* -- a code library for handling user interfaces and the main event loop of a typical application. Most of the code needed for a program is embodied in the application framework. The programmer is responsible for only task-specific code.
- The ABEs are *object editors* for constructing a user interface. From the point of view of the programmer, all that's done is dragging and arranging iconic representations of the GUI elements onto the screen, then defining the methods to be called when these elements are selected by the user. Behind the scenes, the editors automatically create objects from the appropriate ABC classes for you and interconnect these objects so that they intercommunicate properly.

- Each GUI element's *visual editor* not only lets you manipulate all aspects of the appearance of that element (for example, a menu item may or may not be highlighted or checked, it could be displayed in a variety of fonts, sizes or style, or may contain icons, etc.), but also lets you set *behaviors* for that element.
- *Behaviors* are the actions taken when a GUI element is selected by the user. It is defined by a programmer-defined method to be called and the number and types of that method's inputs. Special ABC classes derived from the parent classes **Behavior** and **Specifier** work behind the scenes to communicate between the ABC user interface code and your own program code.
- *Documents* are representations of task-specific program data that may be displayed in a window, saved to a file or printed. The **Document** class and the **Document Editor** mediate these processes, while **Mapping** classes intercommunicate between the document and other classes responsible for the display of data on the screen or on a printed page as well as file access.