31

31 Frameworks for Understanding

It is unlikely that you will learn how to use a "framework class library" in any of your computer science courses at university. Frameworks don't possess quite the social cachet of "normal forms for relational databases", "NP complexity analysis", "LR grammars", "formal derivation of algorithms", "Z specification" and the rest. You might encounter "Design Patterns" (which relate to frameworks) in an advanced undergraduate or graduate level course; but, on the whole, most universities pay little regard to the framework-based development methodologies.

However, if you intend to develop any substantial program on your personal computer, you should plan to use the framework class library provided with your development environment.

As illustrated by the simple "RecordFile" framework presented in the last chapter, a framework class library will embody a model for a particular kind of program. Some of the classes in the framework are concrete classes that you can use as "off the shelf" components (things like the `MenuWindow` and `NumberDialog` classes in the RecordFile framework). More significant are the partially implemented abstract classes. These must be extended, through inheritance, to obtain the classes that are needed in an actual program.

Of course, the framework-provided member functions already define many standard patterns of interactions among instances of these classes. Thus, in the RecordFile framework, most of the code needed to build a working program was already provided by the `Application`, `Document`, and `Record` classes along with helpers like `RecordWindow`.

If you can exploit a framework, you can avoid the need to reimplement all the standard behaviours that its code embodies.

The "typical framework"

The "classic Macintosh application"

The commonly encountered frameworks provide a model for what might be termed "the classic Macintosh application". With some earlier prototypes at Xerox's research center, this form of program has been around since 1984 on the Macintosh (and, from rather later, on Intel-Windows machines). It is the model embodied in

your development environment, in your word processor program, your spreadsheet, your graphics editor, and a host of other packages.

Application, Document, View, and Window

You have an "Application". It creates "Documents" (which can save their data to disk files). The contents of documents are displayed in "Views" inside "Windows". (On the Macintosh, separate documents have completely separate windows. On a Microsoft Windows system, an "Application" that supports a "multiple document interface" has a main window; the individual documents are associated with subwindows grouped within this main window). The same data may appear in multiple views (e.g. a spreadsheet can display a table of numbers in one view while another view shows a pie-chart representing the same values).

"Pick and object, give it a command"

The manuals for these programs typically describe their operation in terms of "picking an object, and giving it a command". The manual writers are thinking of "objects" that are visual elements displayed in the views. In most cases, these will represent actual objects as created and manipulated by the program.

"Picking an object" generally involves some mouse actions (clicking in a box, or dragging an outline etc). The bit about "giving a command" generally refers to use of a menu, or of some form of tool bar or tool palette. For instance, in your word processor you can "pick a paragraph" and then give a command like "set 'style' to 'margin note'"; while in a drawing program you can pick a line and select "Arrows… at start". Such commands change attributes of existing objects. New objects can be created in a variety of ways, e.g. drawing, typing, or menu commands like "paste".

"Network" data structures

Using the editing tools associated with a program, you build an elaborate "network" data structure. Most of the programs will represent their data in memory as a network of separate "objects" interlinked via pointers.

If the program is a word processor, these objects might include "text paragraphs", "pictures", "tables", "style records", "format run descriptors" (these associate styles with groups of characters in paragraphs, they link to their paragraphs via pointers), and so forth. The sequence of "paragraphs" and "pictures" forming a document might be defined by a list.

A draw program will have as basic elements things like lines, boxes, ovals; the basic components of a picture are instances of these classes (each instance having its own attributes like colour, size, position). Individual elements can be grouped; such groups might be represented by lists. Usually, a draw program allows multiple groups, so its primary data structure might be a list of lists of elements. The ordering in the top level list might correspond to the front to back ordering in which items are drawn.

A spreadsheet might use an array of cells whose contents are pointers to data objects like text strings, numbers, and "formulae objects". There would be other data structures associated with the array that help identify relationships amongst cells so that if a number in a cell gets changed, the dependent formulae are recalculated.

This network data structure (or a printout of a visual representation) might be all that the program produces (e.g. word processor and draw programs). In other cases, like the spreadsheet, you want to perform calculations once the data are entered.

Example frameworks

The frameworks that you are most likely to encounter are, for the Macintosh:

- Symantec's Think Class Library (TCL) Version 2
- Apple's MacApp Version 3.5

and for Intel-Windows:

- Borland's Object Windows Library (OWL) Version 2
- Microsoft Foundation Classes (MFC) Version 1.5

Several other companies have competing products for the Intel-Windows environment.

Similar underlying model

There are substantial similarities among these frameworks because they all attempt to model the same type of interactive program and all have been influenced, to varying degrees, by the earlier experimental MacApp 1 and MacApp 2 libraries. (The Borland framework includes a slightly simpler Program-Window model in addition to the " standard" Application-Document-Window-View model. If you are using the Borland framework, you will probably find it worthwhile using the simpler Program-Window model for the first one or two programs that you try to implement.)

A similar class library exists for Unix. This ET++ class library is available free via ftp on the Internet (on Netscape, try ftp://ftp.ubilab.ubs.ch/pub/ET++/). It might be possible to adapt ET++ for use on a Linux environment.

Some C++ limitations

There are "problems" with the C++ code in all these libraries. The Borland library probably makes the best use of the current C++ language. The ET++ library was written several years ago, before C++ supported either multiple inheritance or templates and, in some respects, its coding styles are dated. The Symantec compiler does not yet support recent language features like exceptions (the library fakes exception handling with the aid of various "macros"). The dialect of C++ used for MacApp is slightly non-standard (it had to be changed so that the library could support use from dialects of Pascal and Modula2 as well as C++). The MacApp system is also relatively expensive compared with the other products and its development environment, although having some advantages for large scale projects, is not nearly as convenient to use as the Integrated Development Environments from Symantec, Borland, or Microsoft.

Library structures

The early class libraries (MacApp and ET++), and also MFC, have essentially all their classes organized in a tree structured hierarchy. This style was influenced by earlier work on class libraries for Smalltalk. There are some disadvantages with this arrangement. "Useful" behaviours tend to migrate up to the base class of the hierarchy. The base "TObject" class acquires abilities so that an instance can: transfer itself to/from disk, duplicate itself, notify other dependent objects any time it gets changed, and so forth. All classes in the hierarchy are then expected to implement these behaviours (despite the fact that in most programs `Window` objects never get asked to duplicate themselves or transfer themselves to disk). This is one

the factors that contribute to complexity of the frameworks (like the three hundred member functions for a scroll bar class).

The structure of the "RecordFile" framework, a "forest" of separate trees, is similar to that of the more modern frameworks. These more modern frameworks can still provide abstractions representing abilities like persistence (the ability of an object to transfer itself to/from a disk file). In modern C++, this can be done by defining a pure abstract (or maybe partially implemented abstract) base class, e.g. class `Storable`. Class `Storable` defines the interface for any object that can save itself (functions like `DiskSize()`, `ReadFrom()`, `WriteTo()`). Persistent behaviours can be acquired by a class using multiple inheritance to fold "storability" in with the class's other ancestry.

In the older frameworks, the "collection classes" can only be used for objects that are instances of concrete classes derived from the base "TObject" class. More modern frameworks use template classes for their collections.

The frameworks are evolving. Apart from ET++, each of the named frameworks is commercially supported and new versions are likely to appear. This evolutionary process tends to parallelism rather than divergence. Clever features added in the release of one framework generally turn up in the following releases of the competitors.

Tutorial examples

It will take you at least a year to fully master the framework library for your development environment. But you should be able to get their tutorial examples to run immediately and be able to implement reasonably interesting programs within two or three weeks of framework usage. ET++ provides the source code for some examples but no supporting explanations or documentation. The commercial products all include documented tutorial examples in their manuals. These tutorials build a very basic program and then add more sophisticated features. The tutorial for Borland OWL is probably the best of those currently available.

There is no point in reproducing three or four different tutorials. Instead the rest of this chapter looks at some general aspects common to all the frameworks.

Chapter topics

Resources, code generators, etc

The following sections touch on topics like "Resources", "Framework Macros", the auxiliary code-generating tools (the so called "Wizards" and "Experts"), and "Graphics" programming. There are no real complexities. But together these features introduce a nearly impenetrable fog of terminology and new concepts that act as a barrier to those just starting to look at a framework library.

Persistence

There is then a brief section on support for "persistent objects". This is really a more advanced feature. Support for persistence becomes important when your program needs to save either complex "network" data structures involving many separate objects interlinked via pointer data members, or collections that are going to include instances of different classes (heterogeneous collections). If your framework allows your program to open ordinary `fstream` type input/output files (like the files used in all previous examples), you should continue to use these so long as your data are simple. You should only switch to using the more advanced

forms of file usage when either forced by the framework or when you start needing to save more complex forms of data.

Events

The final sections look at the basic "event handling" structure common to these frameworks, and other common patterns of behaviour.

31.1 "RESOURCES" AND "MACROS"

Resources

"Resource" data structures

Most of the frameworks use "Resources". "Resources" are simply predefined, pre-initialized data structures.

Resources are defined (using "Resource Editors") externally to any program. A file containing several "resources" can, in effect, be include in the list of compiled files that get linked to form a program. Along with the resources themselves, a resource file will contain some kind of index that identifies the various data structures by type and by an identifier name (or unique identifier number).

When a program needs to use one these predefined data structures, it makes a call to the "Resource Manager" (a part of the MacOS and Windows operating systems). The Resource Manager will usually make a copy, in the heap, of the "Resource" data structure and return a pointer to this copy. The program can then do whatever it wants with the copy.

Resources originated with the Macintosh OS when that OS and application programs were being written in a dialect of Pascal. Pascal does not support direct initialization of the elements of an array or a record structure. It requires the use of functions with separate statements that explicitly initialise each of the various individual data elements. Such functions are very clumsy. This clumsiness motivated the use of the separate "resource" data structures. These were used for things like arrays of strings (e.g. the words that appear as menu options, or the error messages that get displayed in alert box), and simple records like a "rectangle" record that defines the size and position of a program's main window.

It was soon found that resources were convenient in other ways. If the error messages were held in an external data structure, rewording of a message to make it clearer (or translation into another language) was relatively easy. The resource editor program would be used to change the resource. There was no need to change program source text and laboriously recompile and relink the code.

All data types – sound, graphics, text, windows

New resource types were defined. Resources were used for sound effects, pictures, fonts and all sorts of other data elements needed by programs. Resource bitmap pictures could be used to define the iconic images that would represent programs and data files. An "alert" resource would define the size and content of a window that could be used to display an error message. A "dialog" resource could define the form of a window, together with the text and number fields used for input of data.

The very first resources were defined as text. The contents of such text files would be a series of things very like definitions of C structs. The files were "compiled" (by a simplified C compiler) to produce the actual resource files.

Resource editors

Subsequently, more sophisticated resource editors have been created. These resource editors allow a developer to chose the form of resource that is to be built and then to enter the data. Textual data can be typed in; visual elements are entered using "graphics editor" components within the resource editor itself. These graphics editors are modelled on standard "draw" programs and offer palettes of tools. An "icon editor" component will have tools for drawing lines, arcs, and boxes; a "window editor" component will have tools that can add "buttons", "text strings", "check boxes" and so forth. Depending on the environment that you are working with, you may have a single resource editor program that has a range of associated editor components, one for each type of resource; alternatively, you may have to use more than one resource editor program.

Macintosh resource editors generally produce resources in "compiled form" (i.e. the final resource file, with binary data representing the resources, is generated directly). The resource editors for the Windows development environments create a text file representation that gets compiled during the program "build" process. You can view the Windows resources that you define in the text format although you will mostly use the graphically oriented components of the resource editor. There are auxiliary programs for the Macintosh development environments that convert compiled resources into a textual form; it would be unusual for you to need to use these.

"Shared" resources for Windows OS

The Windows development environments allow resources to be "shared". A resource file that you create for a new project can in effect "#include" other resource files. This has the advantage of saving a little disk space in that commonly used resources (e.g. the icons that go in a toolbar to represent commands like "File/Open", or "Print") need only exist in some master resource file rather than being duplicated in the resource files for each individual project. However there are disadvantages, particularly for beginners. Frequently, a beginner has the opinion that any resource associated with their project can be changed, or even deleted. Deleting the master copy of something like the main "file dialog" resource causes all sorts of problems. If you are working in a Windows development environment, check the text form of a resource file to make sure the resource is defined in the file rather than #included. The resource editor will let you duplicate a resource so that you can always get a separate, and therefore modifiable copy of a standard resource.

The individual resources of a given type have identifier numbers (e.g. `MENU 1001`, `MENU 1002`, etc). These need to be used in both the resource file and the program text where the resources are used. If the numbers used to identify particular resources are inconsistent, the program won't work. In the Macintosh environments, you just have to keep the numbers consistent for yourself.

In the Windows environments, you can use a header file that contains a series of `#define` constants (e.g. `#define MYFILEMENU 1001`). This header file can be #included in both the (source text) resource file and in program (.cp) files. This reduces the chance of inconsistent usage of resource numbers. Instead of a separate header file, you can put such declarations are the start of the resource (.rc) file. Any resource #includes and actual definitions of new resources can then be bracketed by C++ compiler directive (e.g. `#ifdef RESOURCEOMPILER … #endif`) that hide these data from the C++ compiler. The resource (.rc) file may then get

#included in the .cp source text files. You will see both styles (separate header and direct #include of a resource .rc file) in your framework's examples.

Resources and classes

Resources were invented long before development switched to C++ and OO programming approaches. Resources are just simple data structures. The frameworks will define classes that correspond to many of the resource types. For example, you might have a program that needs bitmap iconic pictures (or multicoloured pixmap pictures); for these you would use instances of some class "BitMapPicture" provided by the framework. The framework's "BitMapPicture" class will have a member function, (e.g. `void BitMapPicture::GetResource( int idnum)`) that allows a `BitMapPicture` object to load a `BITMAP` resource from a file and use the resource data to initialize its own data fields. On the whole, you won't encounter problems in pairing up resource data structures and instances of framework classes.

However, there can be problems with some forms of "window" resources (these problems are more likely to occur with the Macintosh development environments as the Windows environments handle things slightly differently). You may want to define a dialog window that contains "static text" strings for prompts, "edit text" strings for text input fields, and "edit number" components for entering numeric data. Your resource editor has no problem with this; it can let you design your dialog by adding `EditText` and `EditNum` components selected from a tools palette. The data structure that it builds encodes this information in a form like an instruction list: "create an `EditNum` and place it here in the window, create an `EditText` and place it here".

The corresponding program would use an instance of some framework class "DialogWindow". Class `DialogWindow` might have a constructor that simply creates an empty window. Its `GetResource()` function would be able to load the dialog window resource and interpret the "instruction list" so populating the new window with the required subwindows. It is at this point that there may be problems.

The basic problem is that the program cannot create an instance of class `EditText` if the code for this class is not linked. Now the only reference to class `EditText` in the entire program may be the implicit reference in something like a resource data structure defining an "InputFileDialog". The linker may not have seen any reason to add the `EditText` code to the program.

Missing component, consult developer!

You can actually encounter run-time errors where an alert box pops up with a message like "Missing component, consult developer". These occur when the program is trying to interpret a resource data structure defining a window that contains a subwindow of a type about which the program knows nothing.

Your framework will have a mechanism for dealing with this possible problem. The exact mechanism varies a lot between frameworks. Generally, you have to have something like an "initialization function" that "names" all the special window related classes that get used by the program. This initialization function might have a series of (macro) calls like "`ForceReference(EditText); ForceReference( EditNum)`" (which ensure that the compiled program would have the code necessary to deal with `EditText` and `EditNum` subwindows).

"Outdated" resource types

You may find that there are different kinds of resource that seem to do more or less the same thing. For example on the Macintosh you will find a "`view`" resource

type and a "DLOG" (dialog) and its related "DITL" (dialog item list) resource types. They may seem to allow you do the same thing – build a dialog window with subwindows. They do differ. One will be the kind of resource that you use with your framework code, the other (in this case the DLOG and DITL resources) will be an outdated version that existed for earlier Pascal based development systems. Check the examples of resource usage that come with the tutorials in your IDE manuals and use the resources illustrated there.

Macros

A "macro" is a kind of textual template that can be expanded out to give possibly quite a large number of lines of text. Macros may have "arguments"; these are substituted into the expansion text at appropriate points. Macros can appear in the source text of programs. They are basically there to save the programmer from having to type out standard code again and again.

For example, you could define the following macro:

Macro definition

```
#define out(o, x) o.write((char*)&x, sizeof(x))
```

This defines a macro named "out". The `out` macro takes two arguments, which it represents by 'o' and 'x'. Its expansion says that a call to "out" should be replaced by code involving a call to the `write()` function, involving a type cast, an address operator and the `sizeof()` operator.

Macro call

This would allow you to use "macro calls" like the following:

```
out(ofile, fMark1);
out(ofile, fStudentName);
```

Macro expansion

These "macro calls" would be "expanded" before the code was actually seen by the main part of the C++ compiler. Instead of the lines `out(ofile, fMark1)` etc the compiler would see the expansions:

```
ofile . write ( ( char * ) & fMark1, sizeof ( fMark1 ) );
ofile . write ( ( char * ) & fStudentName, sizeof (
fStudentName) );
```

Framework defined macros

The frameworks, particularly ET++ and those for the Windows environment, require lots of standard code to be added to each program-defined class that is derived from a framework-defined class. Macros have been provided so as to simplify the process of adding this standard code.

Declare and define macros

Most of these framework macros occur in pairs. There will be a macro that must be "called" from within the declaration of a class, and another macro that must be called somewhere in the file containing the implementation code for that class. The first macro would declare a few extra member functions for the class; the second would provide their implementation.

For example, you might have something like the following:

- a "DECLARE_CLASS" macro that must go in the class declaration and must name the parent class(es)

- a "DEFINE_CLASS" macro that must be in the implementation.

If you wanted class `MyDoc` based on the framework's `Document` class, you might have to have something like the following:

Example

```
class MyDoc : public Document {
    DECLARE_CLASS(MyDoc, Document)
    …
};
```

with the matching macro call in the implementation:

```
// MyDoc class

DEFINE_CLASS(MyDoc)

Record *MyDoc::MakeEmptyRecord()
{
    …
}
```

If you are curious as to what such macros add to your code, you can always ask the compiler to stop after the macro preprocessing stage. You can then look at the expanded text which will contain the extra functions. A typical extra function would be one that returns the class's name as a text string. You don't really need to know what these functions are; but because they may get called by the framework code they have to be declared and defined in your classes.

Check the macros that you must use

Your framework's documentation will list (but possibly not bother to explain) the macros that you must include in your class declarations and implementation files. If you use the "Class Expert/Wizard" auxiliary programs (see next section) to generate the initial outlines for your classes, the basic macro calls will already have been added (you might still need to insert some extra information for the calls).

31.2 ARCHITECTS, EXPERTS, AND WIZARDS

If you look at the code for the example "StudentMarks" program in Chapter 30, you will see that some is "standard", and much of the rest is concerned only with the initial construction and subsequent interaction with a display structure.

Every application built using a framework needs its own specialized subclasses of class `Application` and class `Document`. Although new subclasses are defined for each program, they have very much the same form in every program. The specialized `Application` classes may differ only in their class names (`LoanApp` versus `StudentRecApp`) and in a single `DoMakeDocument()` function. The `Document` classes also differ in their names, and have different additional data members to store pointers to program specific data classes. But again, the same

group of member functions will have to be implemented in every specialized `Document` class, and the forms of these functions will be similar.

Work with display structures is also pretty much stereotyped. The construction code is all of the form "create an XX-subwindow and put it here". The interaction code is basically "Which subwindow? Get data from chosen window.".

Such highly stereotyped code can be generated automatically.

Automatic code generation

The "Architects", "Experts", or "Wizards" that you will find associated with your Integrated Development Environment are responsible for automatically generating the initial classes for your program and, in association with a resource editor, for simplifying the construction of any interactive display structures used by your program.

Symantec's Visual Architect packages all the processes that you need to generate the initial project with its code files, and interface resources. In the two Windows environments, these processes are split up. The Application Wizard (Expert) program generates all the basic project files. The Resource Workshop (or App Studio) resource editor program is used to add resources. The Class Wizard (Expert) program can be used to help create code for doing things like responding to a mouse-button click in an "action-button", or handling the selection of an option from a displayed menu.

Project generation

The project generation process may involve little more than entry of the basename for the new project and specification of a few parameters. The basename is used when choosing the names for specialized `Application` and `Document` classes; for example, the basename "My" would typically lead to the generation of a skeletal `class MyApp : public Application { }` and a skeletal `class MyDocument : public Document { }`. Some of the parameters that have to be entered in dialogs relate to options. For example, one of the dialogs presented during this project generation process might have a "Supports printing" checkbox. If this is checked, the menus and toolbars generated will include the standard print-related commands, and provision will be made for linking with the framework code needed to support printing. Other parameters define properties like the Windows' memory model to be used when code is compiled. (The Windows OS basically has a choice of 16-bit or 32-bit addressing.) When confronted with a dialog demanding some obscure parameter, follow the defaults specified in the framework's reference manuals.

In the Windows environments, once you have entered all necessary information into the dialogs, the Application Wizard/Expert completes its work by generating the initial project files in a directory that you will have specified. These generated files will include things equivalent to the following:

- a resource file (.rc) which will contain a few standard resources (maybe just a File/Quit menu!);
- a main.cp file which will contain the "main" program (naturally this has the standard form along the lines "*create a MyApp object and tell it to run*");

- MyApp.h, MyApp.cp, MyDocument.h, MyDocument.cp; these files will contain the class declarations and skeletal implementations for these standard classes;

- "build" and/or "make" file(s) (.mak); these file(s) contain information for the project management system including details like where to find the framework library files that have to be linked when the program gets built.

(Symantec's Visual Architect does not generate the project files at this stage. Eventually it creates a rather similar set of files, e.g. a .rsrc file with the resources, a main.cp, and so forth. The information equivalent to that held in a .mak file is placed in the main project file; it can be viewed and changed later if necessary by using one of the project's menu options.)

The MyApp.h, MyApp.cp files now contain the automatically generated text. The contents would be along the following lines:

Automatically generated MyApp.h

```
// Include precompiled framework header
#include "Framework.h"

class MyApp : public Application {
    DECLARE_CLASS(MyApp, Application)
public:
    virtual void  DoSplashScreen();
    virtual void  DoAbout();
    virtual Document *DoMakeDocument();
};
```

Automatically generated MyApp.cp

```
#include "MyApp.h"
#include "MyDocument.h"

DEFINE_CLASS(MyApp)

void MyApp::DoSplashScreen()
{
    // Put some code here if you want a 'splash screen'
    // on start up
}

void MyApp::DoAbout()
{
    // Replace standard dialog with a dialog about
    // your application
    Dialog d(kDEFAULD_ABOUT);  // The default dialog
    d.PoseModally();
}

Document *MyApp::DoMakeDocument()
{
    return new MyDocument;
}
```

The generated code will include effective implementations for some functions (e.g. the `DoMakeDocument()` function) and "stubs" for other functions (things like

the `DoSplashScreen()` function in the example code shown). These stub routines will be place holders for code that you will eventually have to add. Instead of code, they will just contain some comments with "add code here" directives.

Including headers

As noted in Chapter 30, you typically have to include numerous header files when compiling the code for any classes that are derived from framework classes or code that uses instances of framework classes. The framework may provide some kind of master header file, e.g. "Framework.h", which #includes all the separate header files.

Precompiled headers

This master header file may be supplied in "precompiled" form.

It takes a lot of time to open and read the contents of fifty or more separate header files and for the compiler to interpret all the declarations and add their information to its symbol table. A precompiled header file speeds this process up. Essentially, the compiler has been told to read a set of header files and construct the resulting symbol table. It is then told to save this symbol table to file in a form that allows it to be loaded back on some future occasion. This saved file is the "precompiled header". Your IDE manuals will contain a chapter explaining the use of precompiled headers.

User interface components

It is possible to build display structures in much the same way as was done in the example in the previous chapter. But you have to go through an entire edit-compile-link-run cycle in order to see the display structure. If one of the components is not in quite the right position, you have to start over and again edit the code.

It is much easier if the user interface components are handled using resources. The resource editor builds the data structure defining the form of the display. This data structure gets loaded and interpreted at run-time, with all the subwindows being created and placed at the specified points. Instead of dozens of statements creating and placing individual subwindows, the program code will have just a couple of lines like `RecordWindow rw; rw->GetResource(1001)`.

The resource editor used to build the views required in dialogs will allow you to move the various buttons, check boxes, edit-texts and other components around until they are all correctly positioned. The editor may even simulate the behaviour of the dialog so that you can do things like check the behaviour of "clusters of radio buttons" (if you select one radio button, the previously selected button should be deselected).

You will use the resource editor just after the initial project generation step in order to build the main windows and any dialogs that you expect to use in your program.

Command numbers associated with controls

When you add an interactive element (button, checkbox, edit-text etc) to a window you will usually be prompted for an associated "command number" (and, possibly, an identifying name). These command numbers serve much the same role as the window identifiers did in the RecordFile framework. When a control gets activated, a report message gets generated that will have to be handled by some other object. These report messages use the integer command numbers to indicate

what action is being requested. The process is somewhat similar to that by which changes to the `EditNum` subwindows in the StudentMarks program got reported to the `StudentRec` object so that it could collect the new data values.

Associating commands and command handlers

The next stage in developing the initial outline for a program typically involves establishing connections between a command number associated with a visual control (or menu item) and the object that is to respond to the request that this command number represents.

In the Windows environments, this is where you use the Class Expert (or Class Wizard). In the Symantec environment, this is just another aspect of using the Visual Architect.

The ideas of command handling are covered in more detail in Section 31.5. As explained there, the operating system and framework will work together to convert user actions (like key presses and mouse button clicks) into "commands" that get routed to appropriate "command handler objects". Commands are represented by little data structures with an integer command number and sometimes other data.

Example of Commands and Command Handlers

If you were using a framework to build something like the StudentMarks program, you would have the following commands to deal with: 1) New Document, 2) Open Document, 3) Quit, 4) New Record, 5) Delete Record, 6) ViewEdit record, 7) Change name, 8) Change assignment 1, …. You would have to arrange that the `Application` object dealt with commands 1…3; that the current `Document` object dealt with commands 4…6, and the current `Record` object dealt with the remainder.

Usually, you would need a separate member function in a class for each command that instances of that class have to deal with. So, the `Document` class would have to have `NewRecord()`, `DeleteRecord()` and `ViewEditRecord()` member functions . There would also have to be some mechanism to arrange that a `Document` object would call the correct member function whenever it received a command to handle.

Dispatching a command to the class's handler function

Once a command has reached the correct instance of the appropriate class it could be dispatched to the right handler routine using a function like the following:

```
void StudentMarksDoc::HandleCommand(int commandnum)
{
    switch(commandnum) {
case cNEW_REC: NewRecord(); break;
    …
}
```

This was basically the mechanism used in the RecordFile framework example and is essentially the mechanism used in Symantec's Think Class Library.

In the Windows environment, the idea is much the same but the implementation differs. Instead of a `HandleCommand()` member function with a `switch` statement, a Windows' "command handler" class will define a table that pairs command

numbers with function identifiers. You can imagine a something like the following:

```
Dispatch_table StuMarkDocTbl[] = {
    { cNEW_REC, StudentMarkDoc::NewRecord }
    { cDEL_REC, StudentMarkDoc::DeleteRecord }
    { cVIEW_REC, StudentMarkDoc::ViewEditRecord }
};
```

Dispatch by table lookup

(It is not meant to be syntactically correct! It is just meant to represent an array whose entries consist of the records pairing command numbers and function names.) There will be code in the framework to search through such an array checking the command numbers; when the number matching the received command is found, the corresponding function gets called.

Now a substantial part of the code for these command handling mechanisms can be standardized. Stub functions like:

```
void StudentMarkDoc::NewRecord()
{
    // Put some code here
}
```

and the `HandleCommand()` function (or the corresponding table for a Windows program) can all be produced once the generator has been told the command numbers and the function names.

The "Class Expert" (or equivalent) will have a set of dialogs that you use to select a command number, pick the class whose instances are to handle that command, and name the function that will handle the command. (For many standard Windows commands, the name of the handler function is predefined.)

Once these data have been entered, the "Class Expert" will add the extra code with function declarations, and stub definitions to the appropriate files. Macros are used for the function dispatch tables; again, there are separate `DECLARE` and `DEFINE` macros. If you looked at the contents of the files at this stage you would find something along the following lines:

StuDoc.h after Class Expert has run

```
// Include precompiled framework header
#include "Framework.h"

class StudentMarkDoc : public Document {
    DECLARE_CLASS(StudentMarkDoc, Document )
public:
    …
protected:
    void   NewRecord();
    void   DeleteRecord();
    void   ViewEditRecord();
    …

    DECLARE_RESPONSE_TABLE(StudentMarkDoc)

};
```

StuDoc.cp after Class Expert has run

```
#include "StuDoc.h"
#include "StudentRec.h"

DEFINE_CLASS(StudentMarkDoc)

DEFINE_RESPONSE_TABLE(StudentMarkDoc, Document)
    COMMAND(cNEW_REC, NewRecord)
    COMMAND(cDEL_REC, DeleteRecord)
    COMMAND(cVIEW_REC, ViewEditRecord)
END_RESPONSE_TABLE

void StudentMarkDoc::NewRecord()
{
    // Put some code here
}
```

In the Symantec Visual Architect, this is the point where all the project specific files get created and then all the framework source files are copied into the project.

A Program that does Nothing – but does everything right

The code that is generated can be compiled and will run. The main window will be displayed, a File/Quit menu option will work, a default "About this program …" dialog can be opened. Your program may even be able to read and write empty files of an appropriate type (file types are specified in the Application Expert phase of this generation process).

It does all the standard things correctly. But of course all the real routines are still just stubs:

```
void StudentMarkDoc::NewRecord()
{
    // Put some code here
}
```

You must now start on the real programming work for your project.

31.3 GRAPHICS

The programs that you build using the frameworks work with the "windows" supported by the underlying operating system (MacOS, Windows, or X-terminal and Unix). Output is all done using the OS graphics primitives (of course filestreams are still used for output to files).

Library of graphics functions

The OS will provide a "graphics library" (strictly, the X-lib graphics library is not part of the Unix OS). This will be a function library with one hundred or more output related functions. These will include functions for drawing lines, arcs, rectangles, individual characters, and character strings.

Because these graphics functions are provided by the OS and not by the development environment, they may not be listed in the IDE manuals. You can get a list of the functions by opening the appropriate header file (Quickdraw.h on the Macintosh, Windows.h on Windows). The environment's help system may contain descriptions of the more frequently used graphics functions. Examples illustrating the use of the graphics library are not normally included in the manuals for the development environment. There are however many books that cover basic graphics programming for Windows or for Macintosh (the ET++ library uses graphics functions similar to those of the Macintosh, internally these are converted into calls to X-lib functions).

The basic graphics calls shouldn't in fact cause much problem; they are all pretty much intuitive. The function library will have functions like the following:

```
void MoveTo(short h, short v);
void Move(short dh, short dv);
void LineTo(short h, short v);
void FrameRect(const Rect *r);
void PaintRect(const Rect *r);
void PaintOval(const Rect *r);
void FrameArc(const Rect *r, short startAngle,
    short arcAngle);
void DrawChar(short ch);
void DrawText(const void *textBuf, short firstByte,
    short byteCount)
```

(If you look in the header files, you may encounter "extern Pascal" declarations. These are actually requests to the C++ compiler to put arguments on the stack in "reverse" order, as was done by a Pascal compiler. This style is largely an historical accident relating to the earlier use of Pascal in both Macintosh and Windows OSs.)

Although the basic functions are quite simple to use, there can be complications. These usually relate to the maintenance of collections of "window attributes".

Window attributes

Each window displayed on the screen has many associated attributes. For example, a window must have a defined depth (1-bit for black and white, or 8-bit colour, or 16-bit colour etc). There will be default foreground (black) and background (white) colours. There will be one or more attributes that define how bitmaps get copied onto the screen; these "mode" (or "brush" or "pen") attributes allow the program to chose whether to simply copy an image onto the screen or to use special effects like using the pattern of bits in an image to invert chosen bits already displayed on the screen. Other attributes will define how the characters are to be drawn; these attributes will include one that defines the "font" (e.g. 'Times', 'Helvetica' – different fonts define differently shaped forms for the letters of the normal "Roman" alphabet and, possibly, define different special characters), another attribute will specify the size of characters. There will be attribute to specify the thickness of lines. Still another will define the current x, y drawing point.

Ports, Graphics Contexts, Device Contexts

These attributes are normally held in a data structure, the form of which is defined by the graphics library. On the Macintosh, this will be a `Port` data

structure; the X-lib library refers to it as a "Graphics Context"; in the Windows environments it is a "Device Context".

Macintosh and X programmers don't have any real problems with these structures. A new `Port` (or equivalent) gets allocated for each window that gets opened. A program can change a `Port`'s attributes as needed. For Windows programmers, things can sometimes get a bit complex. The Windows OS preallocates some graphics related data structures; these must be shared among all programs. Device context structures (normally accessed via framework defined "wrapper" classes) may have to use shared information rather than allocating new structures. Developers working on Windows will at some stage have to supplement their IDE manuals with a textbook that provides a detailed presentation of Windows graphics programming.

31.4 PERSISTENT DATA

If the data for your program are something simple like a list of "student records" (as in the example in Chapter 30), you should not have any problems with input and output. The automatically generated code will include stub routines like the following (the function names will differ):

```
void StudentRecDoc::WriteTo(fstream& out)
{
    // Put your code here
}
```

The code that you would have to add would be similar to that illustrated for the `ArrayDoc` class in Chapter 30. You would output the length of your list, then you would iterate down the list getting each `StudentRec` object to write its own data.

The input routine would again be similar to that shown for class `ArrayDoc`. The number of records would be read, there would then be a loop creating `StudentRec` objects and getting them to read themselves. After you had inserted your code, the input function would be something like:

```
void StudentRecDoc::ReadFrom(fstream& in)
{
    // Put your code here
    // OK!
    int len;
    in.read((char*)&len, sizeof(len));
    for(int i = 0; i < len; i++) {
           StudentRec *r = new StudentRec;
           r->ReadFrom(in);
           fList.Append(r);
           }
}
```

You would however have problems if you were writing a program, like that discussed briefly in Chapter 23, where your data was a list of different

`CircuitElements`. If you remember the example, class `CircuitElement` was an abstract class; the objects that you would actually have would be instances of subclasses like class `Battery` and class `Resistor`.

Heterogeneous collection

You could store the various `Battery`, `Resistor`, `Wire` etc objects in a single list. This would be a "heterogeneous collection" (a collection of different kinds of data element). Most of your code would work fine with this collection, including the `WriteTo()` function which could be coded along the following lines:

```
void CircuitDoc::WriteTo(fstream& out)
{
    int len = fList.Length();
    out.write((char*) &len, sizeof(len));
    ListIterator LI((&fList);
    LI.Start();
    while(!LI.IsDone()) {
           CircuitElement* e =
                  (CircuitElement*) LI.CurrentItem();
           e->WriteTo(out);
           LI.Next();
           }
}
```

Input problems!

The problems arise with the input. Obviously, the input routine cannot create `CircuitElement` objects because `CircuitElement` is a pure abstract class. The input routine has to create a `Battery`, or a `Wire`, or whatever else is appropriate for the next data item that is to be read from the file.

Solution: class identifying tokens in the file

If you have to deal with heterogeneous collections, you need some scheme whereby objects that write themselves to file start by *outputting some token that identifies their class*. Then you can have an input routine along the following lines:

```
void CircuitDoc::ReadFrom(fstream& in)
{
    int len;
    in.read((char*)&len, sizeof(len));
    for(int i = 0; i < len; i++) {
           Input of class token
           Create instance 'e' of appropriate class
           e->ReadFrom(in);
           fList.Append(e);
           }
}
```

This business about outputting class tokens when objects write themselves to file, then inputting these tokens and interpreting them so as to create an instance of the correct class, all involves a fair amount of code. But this code is now pretty much standardized.

Framework support

The standard code can be implemented by the framework. It will be called something like the "Streamable" component, or "Persistent" component, or "Storable" component. It requires a few extra functions in each class (e.g. a function to write a "class token", which is normally a string incorporating the class name). The framework may require extra macros in the class, e.g. a

`DECLARE_STREAMABLE` macro call in the class declaration and a matching `DEFINE_STREAMABLE` in the implementation. A class using the mechanism may also have to inherit from some `Streamable` base class. (Details differ between frameworks; read your framework manuals.)

There is a second problem related to input and output of complex structures. This one is actually rather harder to deal with; but, again, a "standard" solution has been found. This standard solution should be built into your framework's code.

Pointers

The problem relates to pointer data members in networks. Figure 31.1 illustrates a simplified version of a "network" data structure for a word processor program. The document has a list of "paragraph" objects. A paragraph has two pointer data members; one links to a "style" object and the other to a block of text. Each paragraph has its own text, but different paragraphs may share the same style object.

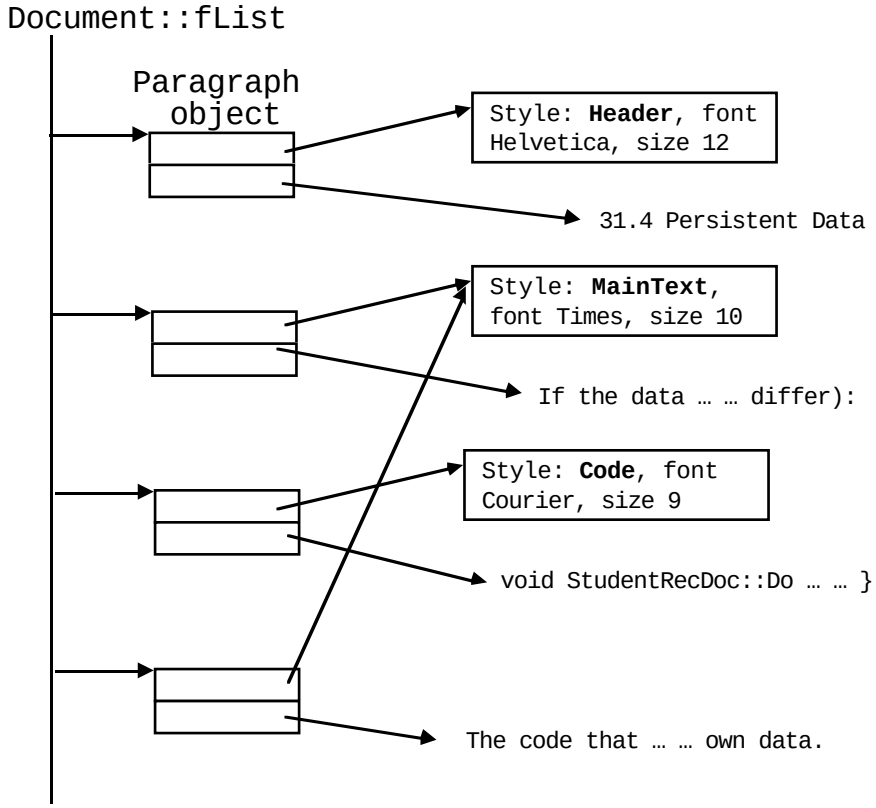

Figure 31.1 Example of "network" data structure with pointers.

You can't have anything like the following code:

```
class Paragraph {
    DECLARE_CLASS(Paragraph)
    DECLARE_STREAMABLE
public:
```

```
    …
    void   WriteTo(fstream& out)
    …
private:
    char   *fText;


    Style  *fStyle;
}
```

This is an incorrect implementation

```
void Paragraph::WriteTo(fstream& out)
{
    Save "Paragraph" class token
    out.write((char*)&fText, sizeof(fText));
    out.write((char*)&fStyle, sizeof(fStyle));
}
```

That `Paragraph::WriteTo()` function does not save any useful data. Two addresses have been sent to a file. Those memory addresses will mean nothing when the file gets reread.

The following is slightly better, but still not quite right:

```
void Paragraph::WriteTo(fstream& out)
{
    Save "Paragraph" class token
    long len = strlen(fText);
    out.write((char*)&len, sizeof(len));
    out.write(fText, len);
    fStyle->WriteTo(out);
}
```

This would save the character data and, assuming `Style` objects know how to save their data, would save the associated styles.

However, you should be able to see the error. The `Style` record named "MainText" will get saved twice. When the data are read back, the second and fourth paragraphs will end up with separate styles instead of sharing the same style record. The word processor program may assume that paragraphs share styles so that if you change a style (e.g. the "MainText" style should now use 11 point) for one paragraph all others with the same style will change. This won't work once the file has been read and each paragraph has its own unshared style record.

What you want is for the output procedure to produce a file equivalent to the following:

```
Object 1: class Paragraph
    20
    31.4 Persistent Data
    Object 2: class Style
           Header Helvetica 12
    End Object 2
End Object 1
Object 3: class Paragraph
    286
    If the data …):
```

```
    Object 4: class Style
           MainText Times 10
    End Object 4
End Object 3
Object 5: class Paragraph
    69
    void … }
    Object 6: class Style
           Code Courier 9
    End Object 6
End Object 5
Object 7: class Paragraph
    The code … data.
    See Object 4
End Object 7
```

When the output has to deal with an object that has already been saved, like the style object for the "MainText" style, a reference back to the earlier copy is all that gets sent to file.

If a file in this form is read, it is possible to reconstruct the original network.

Now getting all of this to work is relatively complicated. It involves modified stream classes that may seem generally similar to the familiar `fstream` classes but which perform additional work. It is these modified streams that know how to deal with pointers and know not to save an object more than once.

The code is pretty much standardized and should be in your framework's "Streamable" component. You will have to read the reference manuals for your framework and the related tutorial examples to see how you are supposed to deal with pointer data members in your objects.

Operator functions

The framework's support for streamability may also include definitions of overloaded operator functions. These will allow you to write code in the form:

```
outfile << fStyle;
```

rather than

```
fStyle->WriteTo(outfile);
```

31.5 THE EVENT HANDLERS

Basics of Event Handling

Programs written for personal computers, and Unix/X-windows, are "event driven".

Events

User actions like keystrokes and mouse clicks are picked up by the operating system. This generates "event" data structures that it queues for processing by the currently running program. Event data structures contain a number of data fields. One will be an integer event code (distinguishing keystroke event, left/right mouse button event etc). The interpretation of the others data fields depends on the event. The data structure for a mouse event will contain x, y coordinate information, while a key event has data to identify which key.

As well as these primary input events, there are others like an "update window" event. The OS will itself generate this type of event. An "update window" event gets sent to a program when the viewable area of an associated window is changed (for instance, another window, possibly belonging to some other program, has been closed letting the user see more of the window associated with the "update" event.

The program picks these events up and interprets them. At the heart of all these programs you will find an event loop like the following:

"Main event loop"

```
do {
    GetNextEvent(anEvent);
    HandleEvent(anEvent);
} while(!fFinished);
```

Operating system's function GetNextEvent()

Function "GetNextEvent()" will be provided by the operating system. It actually switches control from the current program to the operating system. The operating system will check the event queue that it has maintained for the program. If there is an event queued, the OS copies the information into the data structure passed in the call and causes a return from the function call. The program can the continue with the execution of its own "HandleEvent()" function.

If there are no events queued for a program, the call to "GetNextEvent()" will "block" that program. The OS will mark the program as "suspended" and, if there is another program that can run, the OS may arrange for the CPU to start executing that other program. When some input does arrive for the blocked program, the OS arranges for it to resume. (The OS might occasionally generate an "idle event" and return this to an otherwise "blocked" program. This gives the program the chance to run briefly at regular intervals so as to do things like flash a cursor.)

Application:: HandleEvent()

The framework will have this "main event loop" in some function called from `Application::Run()`. The function `Application::HandleEvent()` will perform the initial steps involved in responding to the user's input.

These initial steps involve first sorting out the overall type of event (key, mouse, update, …) and then resolving subtypes. A mouse event might be related to selecting an item from a menu, or it could be a "click" in a background window that should be handled by bringing that window to the front, or it could be a "click" that relates to the currently active window. A key event might signify character input but it might involve a "command" or "accelerator" key, in which case it should really be interpreted in the same way as a menu selection.

This initial interpretation will determine how the event should be handled and identify the object that should be given this responsibility. Sometimes, the handling of the low-level event will finish the process. Functions can return, unwinding the call stack until the program is again in the main event loop asking the OS for the next event.

Example, dealing with a simple "update event"

For example, if the `Application` receives an "update window" event it will also get a "window identifier" as used by the OS. It can match this with the OS window identifiers that it knows are being used by its windows and subwindows, and so determine which of its window objects needs updating. This window object can be sent an "update" or "paint" request. The receiving window should deal with this request by arranging to redraw its contents. Processing finishes when the contents have been redrawn.

Conversion of low-level events to higher level commands

More typically, the objects that are given the low-level events to handle will turn them into "commands". Like "events", these commands are going to be represented by simple data structures (the most important, and only mutually common data field being the identifying command number). "Commands" get handed back to the `Application` object which must route them to the appropriate handler.

MenuManager turns a mouse event into command

For example, if the `Application` object determines that the low level event was a mouse-down in a menu bar, or keying of an accelerator key, it will forward details to a `MenuManager` object asking it to sort out what the user wants. The `MenuManager` can deal with an accelerator key quite easily; it just looks up the key in a table and returns a "command" data structure with the matching command number filled in. Dealing with a mouse-initiated menu selection takes a little more time. The `MenuManager` arranges with the OS for the display of the menu and waits until the user completes a selection. Using information on the selection (as returned by the OS) the `MenuManager` object can again determine the command number corresponding to the desired action. It returns this to the `Application` object in a "command" data structure.

A "control" window turns a mouse event into a command

A mouse click in a control window (an action button, a scroll bar, or something similar) will result in that window object executing its `DoClick()` function (or something equivalent). This will simply create a command data structure, filling in the command number that was assigned to the control when it was built in the resource editor.

The `Application` then has the responsibility of delivering these command data structures (and, also, ordinary keystrokes) to the appropriate object for final processing.

Normally, there will be many potential "handlers" any one of which might be responsible for dealing with a command. For example, the application might have two documents both open; each document will have at least one window open; these windows would contain one or more views. Views, windows, documents and even the application itself are all potentially handlers for the command.

"Command Handler Chain"

The frameworks include code that establishes a "chain of command". This organization simplifies the process of finding the correct handler for a given command.

"First handler", "Target", or "Gopher"

At any particular time, there will be a current "target" command handler object that gets the first chance at dealing with a command. This object will be identified by a pointer data member, `CommandHandler *fTarget`, in the `Application` object. Usually, the target will be a "View" object inside the frontmost window.

The `Application` starts the process by asking the target object to deal with the command (`fTarget->DoCommand(aCommand);`). The target checks to determine whether the command `aCommand` is one that it knows how to deal with. If the target is the appropriate handler, it deals with the command as shown in earlier examples.

Passing the buck

If the target cannot deal with the command, it passes the buck. Each `CommandHandler` object has a data member, `CommandHandler *fNextHandler`, that points to the next handler in the chain. A `View` object's `fNextHandler` will identify the enclosing `Window` (or subwindow). A `Window` that is enclosed within another `Window` identifies the enclosing `Window` as its "next handler". A top level window usually identifies the object that created it (a `Document` or `Application`)

as its next handler. `Document` objects pass the buck to the `Application` that created them.

The original target passes the command on: `fNextHandler->DoCommand( aCommand)`. The receiving `CommandHandler` again checks the command, deals with it if appropriate, while passing on other commands to its next handler.

For example, suppose the command is `cQUIT` (from the File/Quit menu). This would be offered to the frontmost `View`. The `View` might know about `cRECOLOR` commands for changing colours, but not about `cQUIT`. So the command would get back to the enclosing `Window`. `Window` objects aren't responsible for `cQUIT` commands so the command would get passed back to the `Document`. A `Document` can deal with things like `cCLOSE` but not `cQUIT`. So, the command reaches the `Application` object. An `Application` object knows what should happen when a `cQUIT` command arrives; it should execute `Application::DoQuit()`.

Keyed input can be handled in much the same way. If an ordinary key is typed, the `fTarget` object can be asked to perform its `DoKeyPress()` function (or something equivalent). If the target is something like an `EditText`, it will consume the character; otherwise, it can offer it to the next handler. Most `CommandHandler` classes have no interest in keyed input so the request soon gets back to the end of the handler chain, the `Application` object. It will simply discard keyed input (and any other unexpected commands that might arrive).

Defining the initial target

The initial target changes as different windows are activated. Most of the code for dealing with these changes is implemented in the framework. A programmer using the framework has only a few limited responsibilities. If you create a `Window` that contains several `Views`, you will normally have to identify which of these is the default target when that `Window` is displayed. (The user may change the target by clicking the mouse within a different `View`.) You may have to specify your chosen target when you build the display structure in the resource editor (one of the dialogs may have a "Target" field that you have to fill in). Alternatively, it may get done at run time through a function call like `Application::SetTarget(CommandHandler*)`.

Exceptions to the normal event handler pattern

In most situations, an input event gets handled and the program soon gets back to the main event loop. However, drawing actions are an exception.

A mouse down event in a `Window` (or `View`) that supports drawing results in that `Window` taking control. The `Window` remains in control until the mouse button is released. The `Window` will be executing a loop in which it picks up mouse movements. (Operating systems differ. There may be actual movement events generated by the OS that get given to the program. Alternatively, the program may keep "polling the mouse", i.e. repeatedly asking the OS for the current position of the mouse.) Whenever the mouse moves, a drawing action routine gets invoked. (This loop may have to start with some code that allows the `Window` to specify that it wants to "capture" incoming events. When the mouse is released, the `Window` may have to explicitly release its hold on the event queue.)

Drawing routines

Most of this code is framework supplied. The application programmer using the framework has to write only the function that provides visual feedback during the drawing operation and, possibly, a second function that gets called when drawing activity ends. This second function would be responsible for giving details of the

drawn item to the `Document` object so that this can add the new item to its list of picture elements.

CommandHandler classes

The frameworks provide the `CommandHandler` class hierarchy. Class `CommandHandler` will be an abstract class that defines some basic functionality (your framework will use different function names):

```
class CommandHandler : public ? {
    …
public:
    void SetNextHandler(CommandHandler* newnext)
           { fNextHandler = newnext; }
    …
    void DoCommand(…);
    void HandleCommand();
    …
    void   DoKeyPress(char ch)
           { fNextHandler->DoKeyPress(ch); }
protected:
    CommandHandler *fNextHandler;
};
```

The `SetNextHandler()` function is used, shortly after a `CommandHandler` gets created, so as to thread it into the handler chain. Command handling may use a single member function or, as suggested, there may be a function, `DoCommand()`, that determines whether a command can be handled (passing others up the chain) while a second function, `HandleCommand()`, sorts out which specialized command handling function to invoke.

The frameworks have slightly different class hierarchies. Generally, classes `Application`, `Document`, and `Window` will be subclasses of `CommandHandler`. There will be a host of specialized controls (button, scroll bar, check box etc) that are also all derived from `CommandHandler`. There may be other classes, e.g. some frameworks have a `Manager` class that fills much the same role as `Document` (ownership of the main data structures, organization of views and windows) but is designed for programs where there are no data that have to be saved to file (e.g. a game or a terminal emulator program).

Framework handles standard commands

The framework code will handle all the standard commands like "File/New", "File/Save As…". The developer using the framework simply has to provide the effective implementations for the necessary functions like `DoMakeDocument()`.

Application specific command handling added to Document and View classes

All the program specific commands (e.g. a draw program's "add line", "recolor line", "change thickness" commands etc) have to be integrated into the framework's command handling scheme. Most of these extra commands change the data being manipulated; a few may change parameters that control the display of the data. The commands that change the data should generally get handled by the specialized `Document` class (because the `Document` "owns" the data). Those that change display parameters could be handled by a `View` or `Window`.

The Class Expert program (or equivalent) should be used to generate the initial stubs for these additional command handling routines. The developer can then place the effective data manipulation code in the appropriate stub functions.

With the overall flow of control largely defined by the framework, the concerns of the development programmer change. You must first decide on how you want to hold your data. In simple cases, you will have a "list" of data elements; with this list (a framework supplied collection class) being a data member that you add to the document. Then, you can focus in turn on each of the commands that you want to support. You decide how that command should be initiated and how it should change the data. You then add the necessary initiating control or menu item; get the stub routines generated using the "Class Expert"; finally, you fill in these stub routines with real code.

On the whole, the individual commands should require only a small amount of processing. A command handling routine is invoked, the data are changed, and control quickly returns to the framework's main event loop. Of course, there will be exceptions where lengthy processing has to be done (e.g. command "Recalculate Now" on a spreadsheet with 30,000 formulae cells).

31.6 THE SAME PATTERNS IN THE CODE

The frameworks just evolved. The Smalltalk development environments of the early 1980s already provided many of the framework features. The first of the main family of frameworks was MacApp 1.0 (≈1986). It had the low level event handling, conversion of OS events into framework commands, and a command handler hierarchy of a fairly standard form. Subsequently, framework developers added new features, and rearranged responsibilities so as to achieve greater utility.

Recently, some of those who had helped develop the frameworks reviewed their work. They were able to retrospectively abstract out "design patterns". These "design patterns" define solutions to similar problems that frequently reoccur.

Pattern: Chain of Responsibility

The last section has just described a pattern – "Chain of Responsibility". The arrangements for the command handler chain were originally just something necessary to get the required framework behaviour. But you can generalize. The "Chain of Responsibility" pattern is a way of separating the sender of a request from the object that finally handles the request. Such a separation can be useful. It reduces the amount of "global knowledge" in a program (global knowledge leads to rigidity, brittleness, breakdown). It increases flexibility, other objects can be inserted into the chain of responsibility to intercept normal messages when it is necessary to take exceptional actions.

Another pattern, "Singleton", was exploited in Chapter 29 for the `WindowRep` class. The `ADCollection` and `BTCollection` classes of Chapter 29 illustrate the Adapter pattern.

These "design patterns" are a means for describing the interactions of objects in complex programs. The published collections of patterns is a source of ideas. After all, it documents proven ways of solving problems that frequently reoccurred in the frameworks and the applications built on those frameworks.

As you gain confidence by programming with your framework you should also study these design patterns to get more ideas on proven problem solving strategies. You will find the patterns in "Design Patterns: Elements of Reusable Object-Oriented Software" by E. Gamma, R. Helm, R. Johnson, and J. Vlissides (Addison-Wesley).

EXERCISES

1 Use your framework to implement a version of the StudentMarks program. Employ a (framework provided) list or a dynamic array class for the collection.