30

30 Reusable designs

The last chapter illustrated some simple uses of inheritance and polymorphism. It is these programming techniques that distinguish "Object Oriented Programming" from the "object based" (or abstract data type) style that was the main focus of Part IV of this text. These Object Oriented (OO) techniques originated in work on computer simulation of real world systems. The "dungeon game" is a simulation (of an unreal world) and so OO programming techniques are well suited to its implementation.

Although OO programming techniques were originally viewed as primarily applicable to simulations, they have over the last ten years become much more widely utilised. This greater use is largely a consequence of the increased opportunity for "reuse" that OO techniques bring to application development.

Reuse, whether of functions, components, or partial designs, always enhances productivity. If you can exploit reusable parts to handle "standard" aspects of an application, you can focus your efforts on the unique aspects of the new program. You will produce a better program, and you will get it working sooner than if you have to implement everything from scratch.

Approaches to Reuse

Reusable algorithms

Reuse with functions

If you are using the "top down functional decomposition" strategy that was illustrated in Part III, you are limited to reusing standard algorithms; the code in the function libraries implements these standard algorithms. Reusing algorithms is better than starting from scratch. These days, nobody writes their own `sin()` function, they use the version in the maths library. Computer science students are often made to rewrite the standard sorting and searching functions, but professionals use `qsort()` and `bsearch()` (standardized sort and binary search functions that are available in almost every development environment). As noted in Chapter 13, over the years huge libraries of functions have been built up, particularly in science and engineering, to perform standard calculations.

Reusable functions are helpful in the case where all you need to do is calculate something. But if you want to build an interactive program with windows and menus etc, you soon discover problems.

Function libraries for interactive programs

There are function libraries that are intended to be used when building such programs. Multi-volume reference manuals exist to describe them. For example, the Xlib reference manuals define the functions you can used to work with the X-windows interface on Unix. The series "Inside Macintosh" describes how to create windows and menus for the Mac. OS. These books include the declarations of literally hundreds of functions, and dozens of data structures. But these function libraries are very difficult to use.

Limitations of function libraries

The functions defined in these large libraries are disjoint, scattered, inconsistently named. There is no coherence. It is almost impossible to get a clear picture of how to organize a program. Instead you are faced with an arbitrary collection of functions, and the declarations of some types of structures that you have to have as globals. Programs built using just these function libraries acquire considerable entropy (chaotic structure). Each function call takes you off to some other arbitrary piece of code that rapes and pillages the global data structures.

Reusable components

Reusable classes and object based design

The "object based" techniques presented in Part IV give you a better handle on reuse. Class libraries and object based programs allow you to reuse abstract data types. The functions and the data that they operate on are now grouped. The data members of instances of classes are protected; the compiler helps make sure that data are only accessed via the appropriate functions.

Program design is different. You start by identifying the individual objects that are responsible for particular parts of the overall data. You define their classes. Often, you will find that you can reuse standard classes, like the collection classes in Chapters 21 and 24. As well as providing working code, these classes give you a way of structuring the overall program. The program becomes a sequence of interactions between objects that are instances of standard and application specific classes.

Essentially, the unit of reuse has become larger. Programs are built at least in part from reusable components. These reusable components include collection classes and, on Unix, various forms of "widget". (A widget is essentially a class that defines a "user interface" component like a menu or an alert box.)

Reusable patterns of object interactions and program designs

Can we reuse more?

When inheritance was introduced in Chapter 23, it was shown that this was a way of representing and exploiting similarities. Many application programs have substantial similarities in their behaviour; such similarities lead to reusable designs.

Similar patterns of interactions in different programs

You launch a program. Once it starts, it presents you with some form of "file dialog" that allows you to create a new file, or open an existing file. The file is opened. One or more windows are created. If the file existed previously, some portions of its current contents get displayed in these new windows. The system's menu bar gets changed, or additional menus or tool bars appear associated with the new window(s). You use the mouse pointer and buttons to select a menu option and a new "tools palette" window appears alongside the document window. You select a tool from the palette. You use the tool to add data to the document.

The behaviour is exactly the same. It doesn't matter whether it is a drawing program or spreadsheet. The same patterns of behaviour are repeated.

Reusable patterns of interaction?

Object oriented programming techniques provide a way of capturing common patterns of behaviour. These patterns involve standardized interactions between instances of different classes.

Capturing standard patterns of interaction in code

The "opening sequence for a program" as just described would involve interactions between an "application" object, a "document" object, several different "window" objects, maybe a "menu manager" object and several others.

An "opening sequence" pattern could specify that the "application" object handle the initial File/New or File/Open request. It should handle such a request by creating a document object and giving it the filename as an argument to an "OpenNew()" or "OpenOld()" member function. In "OpenOld()", the document object would have to create some objects to store the data from the file and arrange to read the existing data. Once the "open" step is complete, the application object would tell the new document object to create its display structure. This step would result in the creation of various windows.

Much is standard. The standard interactions among the objects can be defined in code:

Default implementation defined

```
Application::HandleCommand( command#, …)
    switch(command#)
newCommand:
           doc = this->DoMakeDocument();
           doc->OpenNew();
           doc->CreateDisplay();
           break;
openCommand:
           filename = this->PoseFileDialog();
           doc = this->DoMakeDocument();
           doc->OpenOld(filename, …);
           doc->CreateDisplay();
           break;
…
```

Default implementation defined

```
Document::OpenOld(filename, …)
    this->DoMakeDataStructures()
    this->DoRead(filename, …)
```

Pure abstract functions, implementation is application specific

```
Document::DoMakeDataStructures() ?

Document::DoRead(…) ?
```

Of course, each different program does things differently. The spreadsheet and drawing programs have to create different kinds of data structure and then have to read differently formatted files of data.

Utilize class inheritance

This is where inheritance comes in.

The situation is very much like that in the dungeon game with class `Monster` and its subclasses. The `Dungeon` code was written in terms of interactions between the `Dungeon` object and instances of class `Monster`. But there were never any `Monster` objects. Class `Monster` was an abstraction that defined a few standard behaviours, some with default implementations and some with no implementation. When the program ran, there were instances of specialized subclasses of class `Monster`; subclasses that owned their own unique data and provided effective implementations of the behaviours declared in class `Monster`.

An abstract class Document

Now, class `Document` is an abstraction. It defines something that can be asked to open new or old files, create displays and so forth. All kinds of document exhibit such behaviours; each different kind does things slightly differently.

Possible specialized subclasses

Specialized subclasses of class `Document` can be defined. A `SpreadSheetDoc` would be a document that owns an array of `Cell` objects where each `Cell` is something that holds either a text label, or a number, or a formula. A `DrawDoc` would be a document that owns a list of `PictureElements`. Each of these specialized subclasses would provide effective definitions for the empty `Document::DoRead()` and `Document::DoMakeDataStructures()` functions (and for many other functions as well!).

Building complete programs

A particular program won't create different kinds of document! Instead, you build the "spreadsheet" program or the "draw" program.

For the "draw" program, you would start by creating class `DrawApp` a minor specialization of class `Application`. The only thing that `DrawApp` does differently is that its version of the `DoMakeDocument()` function creates a `DrawDoc`. A `DrawDoc` is pretty much like an ordinary `Document`, but it has an extra `List` data member (to store its `PictureElements`) and, as already noted, it provides effective implementations of functions like `DoRead()`.

Such a program gets built with much of its basic structure defined in terms of classes that are specializations of standardized, reusable classes taken from a library. These reusable classes are the things like `Application`, `Document`, and `Window`. Some of their member functions are defined with the necessary code in the implementation files. Other member functions may have empty (do nothing) implementations. Still other member functions are pure virtual functions that must be given definitions in subclasses.

Reusing a design

Reusing a design

A program built in this fashion illustrates reuse on a new scale. It isn't just individual components that are being reused. Reuse now extends to design.

Design ideas are embedded in the code of those functions that are defined in the library. Thus, the "standard opening sequence" pattern implements a particular design idea as to how programs should start up and allow their users to select the data files that are to be manipulated. Another defined pattern of interactions might specify how an `Application` object was to handle a "Quit" command (it should first give any open document a chance to save changes, tell the document to close its windows and get rid of data, delete the document, close any application windows e.g. floating tool palettes, and finally quit).

Default code implements the "standard look and feel"

The code given for the standard classes will embody a particular "look and feel" as might be required for all applications running on a particular type of machine. The specifications for a new application would normally require compliance with "standard look and feel". If you had to implement a program from scratch, you would have to sort out things like the "standard opening sequence" and "standard quit" behaviours and implement all the code. If you have a class library that embodies the design, you simply inherit it and get on with the new application specific coding.

Framework class libraries

It is increasingly common for commercial products to be built using standardized framework class libraries. A "framework class library" has the classes that provide the basic structure, the framework, for all applications that comply with a standardized design. The Integrated Development Environment that you have on your personal computers includes such a class library. You will eventually get to use that library.

A simplified example framework

The real framework class libraries are relatively complex. The rest of this chapter illustrates a simplified framework that can serve as an introduction.

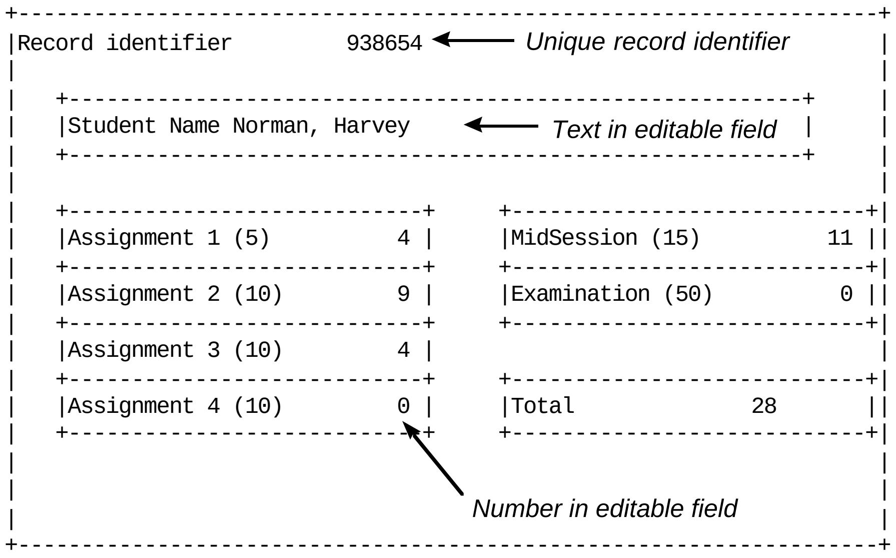
While real frameworks allow for many different kinds of data and document; this "RecordFile" framework is much more restricted. Real frameworks allow for multiple documents and windows; here you make do with just one of each. Real frameworks allow you to change the focus of activity arbitrarily so one moment you can be entering data, the next moment you can be printing some graphic representation of the data. Here, the flow of control is much more predefined. All these restrictions are needed to make the example feasible. (The restrictions on flow of control are the greatest simplifying factor.)

30.1 THE RECORDFILE FRAMEWORK: CONCEPTS

The "RecordFile" framework embodies a simple design for any program that involves updating "records" in a data file. The "records" could be things like the customer records in the example in Chapter 17. It is primarily the records that vary between different program built using this framework.

Example program and record

Figure 30.1 shows the form of the record used in a program, "StudentMarks", built using the framework. This program keeps track of students and their marks in a particular subject. Students' have unique identifier numbers, e.g. the student number 938654. The data maintained include the student's name and the marks for assignments and exams. The name is displayed in an editable text field; the marks are in editable number entry fields. When a mark is changed, the record updates the student's total mark (which is displayed in a non-editable field.)



```
+--------------------------------------------------------------------+
|Record identifier         938654                                    |
|                                                                    |
|   +--------------------------------------------------------+      |
|   |Student Name Norman, Harvey                             |      |
|   +--------------------------------------------------------+      |
|                                                                    |
|   +----------------------------+   +----------------------------+|
|   |Assignment 1 (5)          4 |   |MidSession (15)          11 ||
|   +----------------------------+   +----------------------------+|
|   |Assignment 2 (10)         9 |   |Examination (50)          0 ||
|   +----------------------------+   +----------------------------+|
|   |Assignment 3 (10)         4 |                                 |
|   +----------------------------+   +----------------------------+|
|   |Assignment 4 (10)         0 |   |Total             28        ||
|   +----------------------------+   +----------------------------+|
|                                                                    |
|                                                                    |
|                                                                    |
+--------------------------------------------------------------------+
```

Figure 30.1 A "record" as handled by the "RecordFile Framework".

Starting: "New", "Open", "Quit"

When the "StudentMarks" program is started, it first presents the user with a menu offering the choices of "New (file)", "Open (existing file)", or "Quit". If the user selects "New" or "Open", a "file-dialog" is used to prompt for the name of the file.

Changing the contents of a file

Once a file has been selected, the display changes, see Figure 30.2. It now displays details of the name of the file currently being processed, details of the number of records in the file, and a menu offering various options for adding, deleting, or modifying records.

Handling a "New record" command

If the user selects "New record", the program responds with a dialog that requires entry of a new unique record identifier. The program verifies that the number entered by the user does not correspond to the identifier of any existing record. If the identifier is unique, the program changes to a record display, like that shown in Figure 30.1, with all editable fields filled with suitable default values.

Handling "Delete …" and "View/edit …" commands

If the user picks "Delete record" or "View/edit record", the program's first response is to present a dialog asking for the identifier number of an existing record. The number entered is checked; if it does not correspond to an existing record, no further action is taken.

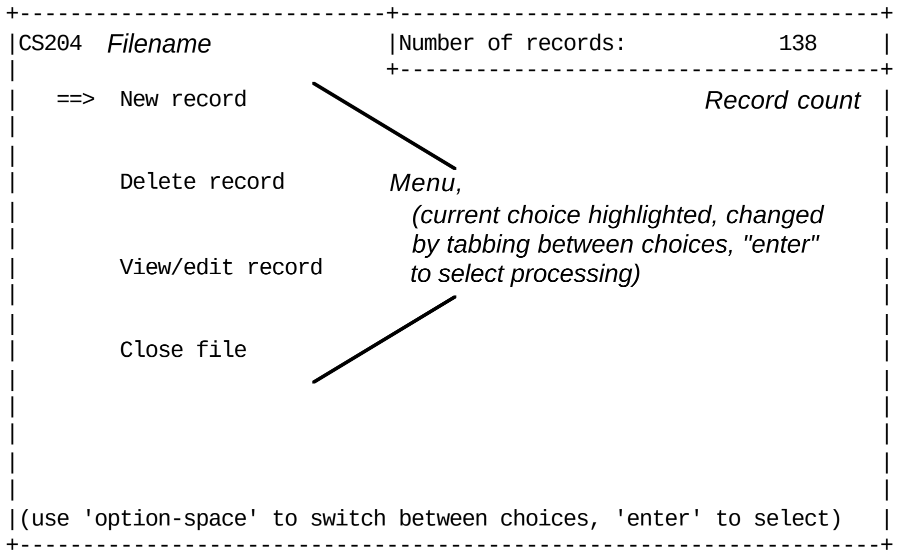
A "Delete record" command with a valid record identifier results in the deletion of that record from the collection. A "View/edit" command leads to a record display showing the current contents of the fields for the record.

Handling a "Close" command

After performing any necessary updates, a "Close" command closes the existing file. The program then again displays its original menu with the options "New", "Open", and "Quit".

Another program, another record display

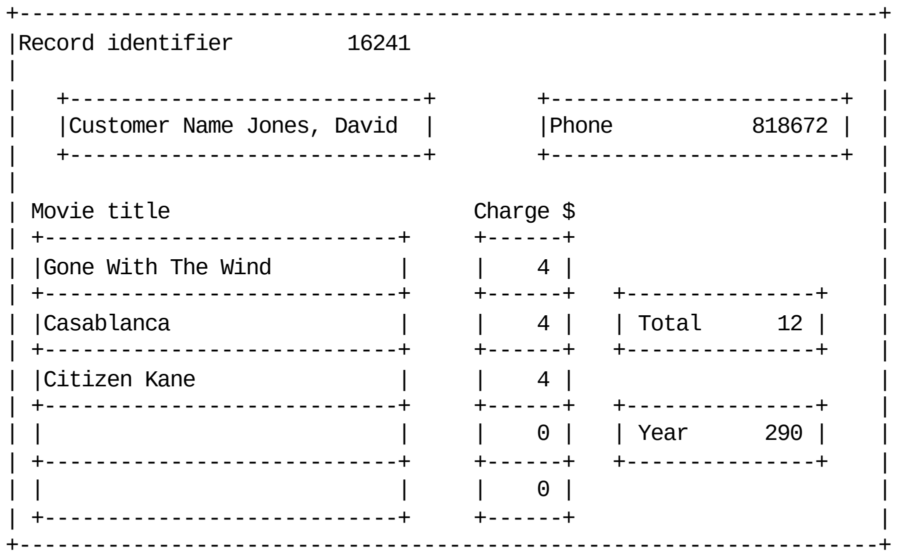
"Loans" is another program built on the same framework. It is very similar in behaviour, but this program keeps track of movies that a customer has on loan from a small video store. Its record is shown in Figure 30.3.



```
+-----------------------------+--------------------------------------+
|CS204                        |Number of records:           138      |
|                             +--------------------------------------+
|   ==>  New record                                                  |
|                                                                    |
|                                                                    |
|        Delete record                                               |
|                                                                    |
|                                                                    |
|        View/edit record                                            |
|                                                                    |
|                                                                    |
|        Close file                                                  |
|                                                                    |
|                                                                    |
|                                                                    |
|                                                                    |
|                                                                    |
|(use 'option-space' to switch between choices, 'enter' to select)  |
+--------------------------------------------------------------------+
```

Filename
Record count
Menu,
(current choice highlighted, changed by tabbing between choices, "enter" to select processing)

Figure 30.2 The menu of commands for record manipulation.



```
+--------------------------------------------------------------------+
|Record identifier         16241                                     |
|                                                                    |
|   +----------------------------+        +-----------------------+  |
|   |Customer Name Jones, David  |        |Phone          818672 |  |
|   +----------------------------+        +-----------------------+  |
|                                                                    |
| Movie title                        Charge $                        |
| +----------------------------+    +------+                         |
| |Gone With The Wind          |    |    4 |                         |
| +----------------------------+    +------+   +---------------+     |
| |Casablanca                  |    |    4 |   | Total      12 |     |
| +----------------------------+    +------+   +---------------+     |
| |Citizen Kane                |    |    4 |                         |
| +----------------------------+    +------+   +---------------+     |
| |                            |    |    0 |   | Year      290 |     |
| +----------------------------+    +------+   +---------------+     |
| |                            |    |    0 |                         |
| +----------------------------+    +------+                         |
+--------------------------------------------------------------------+
```

Figure 30.3 Record from another "RecordFile" program.

The overall behaviours of the two programs are identical. It is just the records that change. With the StudentMarks program, the user is entering marks for different pieces of work. In the Loans program, the user enters the names of movies and rental charges.

30.2 THE FRAMEWORK CLASSES: OVERVIEW

The classes used in programs like "StudentMarks" and "Loans" are illustrated in the class hierarchy diagram shown in Figure 30.4.

Class browser

Most IDEs can produce hierarchy diagrams, like that in Figure 30.4, from the code of a program. Such a diagram is generated by a "class browser". A specific class can be selected, using the mouse, and then menu commands (or other controls) can be used to open the file with the class declaration or that with the definition of a member function chosen from a displayed list. A class declaration or function definition can be edited once it has been displayed. As you get more deeply into the use of classes, you may find the "browser" provides a more convenient editing environment than the normal editor provided by the IDE.

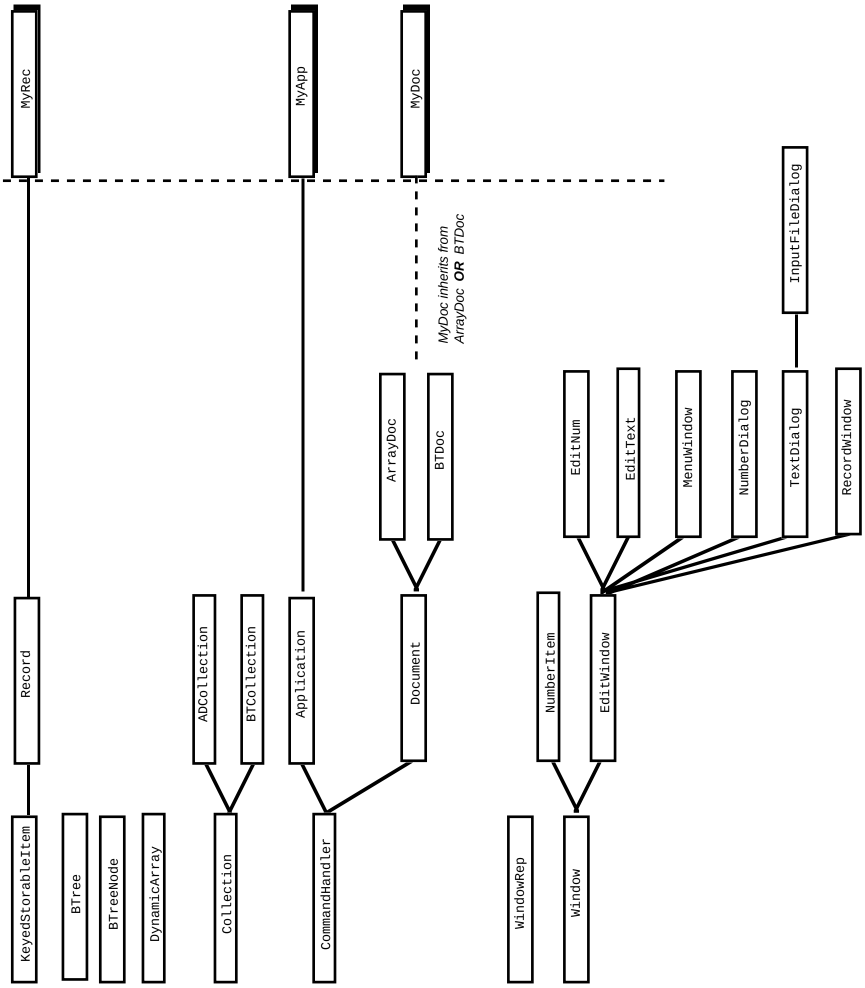


Figure 30.4 Class hierarchy for "RecordFile" Framework.

As illustrated in Figure 30.4, a program built using the framework may need to define as few as three classes. These are shown in Figure 30.4 as the classes `MyApp`, `MyDoc`, and `MyRec`; they are specializations of the framework classes `Application`, `Document`, and `Record`.

Classes KeyedStorableItem, Record, and MyRec

KeyedStorableItem

The class `KeyedStorableItem` is simply an interface (same as used in Chapter 24 for the `BTree`). A `KeyedStorableItem` is something that can report its key value (in this case, the "unique record identifier"), can say how much disk space it occupies, and can be asked to transfer its permanent data between memory and file. Its functions are "pure virtual"; they cannot be given any default definition, they must be defined in subclasses.

Record

Class `Record` adds a number of additional behaviours. `Record` objects have to be displayed in windows. The exact form of the window depends on the specific kind of `Record`. So a `Record` had better be able to build its own display window, slotting in the various "EditText" and "EditNum" subwindows that it needs. Since the contents of the "edit" windows can get changed, a `Record` had better be capable of setting the current value of a data member in the corresponding edit window, and later reading back a changed value. Some of these additional functions will be pure virtual, but others may have "partial definitions". For example, every `Record` should display its record identifier. The code to add the record identifier to the display window can be coded as `Record::AddFieldsToWindow().` A specialized implementation of `AddFieldsToWindow(),` as defined for a subclass, can invoke this standard behaviour before adding its own unique "edit" subwindows.

MyRec

Every specialized program built using the framework will define its own "MyRec" class. (This should have a more appropriate name such as `StudentRec` or `LoanRec`.) The "MyRec" class will define the data members. So for example, class `StudentRec` would specify a character array to hold the student's name and six integer data members to hold the marks (the total can be recomputed when needed). A `LoanRec` might have an array of fixed length strings for the names of the movies on loan.

The specialized `MyRec` class usually wouldn't need to add any extra functionality but it would have to define effective implementations for all those pure virtual functions, like `DiskSize()` and `ReadFrom()`, declared in class `KeyedStorableItem`. It would also have to provide the rest of the implementation of functions like `Record::AddFieldsToWindow()`.

Collection classes and "adapters"

Programs work with sets of records: e.g. all the students enrolled in a course, or all the customers of the video store.

A program that has to work with a small number of records might chose to hold them all in main memory. It would use a simple collection class like a dynamic array, list, or (somewhat better) something like a binary tree or AVL tree. It would work by loading all its records from file into memory when a file was opened, letting the user change these records and add new records, and finally write all records back to the file.

A program that needed a much larger collection of records would use something like a BTree to store them.

Collection classes

The actual "collection classes" are just those introduced in earlier chapters. The examples in this chapter use class `DynamicArray` and class `BTree`, but any of the other standard collection classes might be used. An instance of the chosen collection class will hold the different `Record` objects. Figure 30.4 includes class `DynamicArray`, class `BTree` and its auxiliary class `BTreeNode`.

The different collection classes have slightly different interfaces and behaviours. For example, class `BTree` looks after its own files. A simpler collection based on an in-memory list or dynamic array will need some additional component to look after disk transfers. But we don't want such differences pervading the main code of the framework.

Adapter classes for different collections

Consequently, the framework uses some "adapter" classes. Most of the framework code can work in terms of a "generic" `Collection` that responds to requests like "append record", "delete record". Specialized "adapter" classes can convert such requests into the exact forms required by the specific type of collection class that is used.

ADCollection and BTCollection

Figure 30.4 shows two adapter classes: `ADCollection` and `BTCollection`. An `ADCollection` objection contains a dynamic array; a `BTCollection` owns a `BTree` object (i.e. it has a `BTree*` data member, the `BTree` object is created and deleted by code in `BTCollection`). These classes provide implementations of the pure virtual functions `Collection::Append()` etc. These implementations call the appropriate functions of the actual collection class object that is used to store the data records. The adapter classes also add any extra functions that may be needed in association with a specific type of collection.

Command Handlers: Application, Document and their subclasses

Although classes `Application` and `Document` have quite different specific roles there are some similarities in their behaviour. In fact, there are sufficient similarities to make it worth introducing a base class, class `CommandHandler`, that embodies these common behaviours.

A `CommandHandler` is something that has the following two primary behaviours. Firstly it "runs". "Running" means that it builds a menu, loops handling commands entered via the menu, and finally tidies up.

CommandHandler ::Run()

```
void CommandHandler::Run()
{
    this->MakeMenu();
    …
    this->CommandLoop();
    …
    this->Finish();
}
```

CommandHandler ::CommandLoop()

The second common behaviour is embodied in the `CommandLoop()` function. This will involve menu display, and then processing of a selected menu command:

```
void CommandHandler::CommandLoop()
{
    while(!fFinished) {
            …
            int c = pose menu dialog …
            this->HandleCommand(c);
            }
}
```

A `CommandHandler` object will continue in its command handling loop until a flag, `fFinished`, gets set. The `fFinished` flag of an `Application` object will get set by a "Quit" command. A `Document` object finishes in response to a "Close" command.

Application

As explained in the previous section, the `Application` object will have a menu with the choices "New", "Open" and "Quit". Its `HandleCommand()` function is:

Application:: HandleComand()

```
void Application::HandleCommand(int cmdnum)
{
    switch(cmdnum) {
case cNEW:
            fDoc = this->DoMakeDocument();
            fDoc->DoInitialState();
            fDoc->OpenNew();
            fDoc->Run();
            delete fDoc;
            break;
case cOPEN:
            fDoc = this->DoMakeDocument();
            fDoc->DoInitialState();
            fDoc->OpenOld();
            fDoc->Run();
            delete fDoc;
            break;
case cQUIT:
            fFinished = 1;
            break;
            }
}
```

The "New" and "Open" commands result in the creation of some kind of `Document` object (obviously, this will be an instance of a specific concrete subclass of class `Document`). Once this `Document` object has been created, it will be told to open a new or an existing file, and then it will be told to "run".

The `Document` object will continue to "run" until it gets a "Close" command. It will then tidy up. Finally, the `Document::Run()` function, invoked via `fDoc->Run()`, will return. The `Application` object can then delete the `Document` object, and resume its "run" behaviour by again displaying its menu.

class MyApp

How do different applications vary?

The application objects in the "StudentMarks" program and "Loans" program differ only in the kind of `Document` that they create. A "MyApp" specialized subclass of class Application need only provide an implementation for the DoMakeDocument() function. Function Application::DoMakeDocument() will be "pure virtual", subclasses must provide an implementation. A typical implementation will be along the following lines:

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

A StudentMarkApp would create a StudentMarkDoc while a LoanApp would create a LoanDoc.

Specialized subclasses of class Application could change other behaviours because all the member functions of class Application will be virtual. But in most cases only the DoMakeDocument() function would need to be defined.

class Document

Class Document is substantially more complex than class Application. It shares the same "run" behaviour, as defined by CommandHandler::Run(), and has a rather similar HandleCommand() function:

Document::HandleCommand()

```
void Document::HandleCommand(int cmdnum)
{
    switch(cmdnum) {
case cNEWREC:
           DoNewRecord();
           break;
case cDELREC:
           DoDeleteRecord();
           break;
case cVIEW:
           DoViewEditRecord();
           break;
case cCLOSE:
           DoCloseDoc();
           fFinished = 1;
           break;
           }
}
```

Functions like DoNewRecord() are implemented in terms of a pure virtual DoMakeRecord() function. It is this Document::DoMakeRecord() function that gets defined in specialized subclasses so as to create the appropriate kind of Record object (e.g. a StudentRec or a LoanRec).

Document objects are responsible for several other activities. They must create the collection class object that they work with. They must put up dialogs to get file names and they may need to perform other actions such as getting and checking record numbers.

Document hierarchy

While the "adapter" classes can hide most of the differences between different kind of collection, some things cannot be hidden. As noted in the discussion above on ADCollection and BTCollection, there are substantial differences between those collections that are loaded entirely into memory from file as a program starts and those, like the `BTree` based collection, where individual records are fetched as needed.

ArrayDoc and BTDoc

There has to be a kind of parallel hierarchy between specialized collection classes and specialized `Document` classes. This is shown in Figure 30.4 with the classes `ArrayDoc` and `BTDoc`. An `ArrayDoc` object creates an instance of an `ADCollection` as its `Collection` object while a `BTDoc` creates a `BTCollection`. Apart from `DoMakeCollection()` (the function that makes the `Collection` object), these different specialized subclasses of `Document` differ in their implementations of the functions that deal with opening and closing of files.

MyDoc

Different programs built using the framework must provide their own specialized `Document` classes – class `StudentMarkDoc` for the StudentMarks program or `LoanDoc` for the Loans program. Figure 30.4 uses class `MyDoc` to represent the specialized `Document` subclass needed in a specific program.

Class `MyDoc` won't be an immediate subclass of class `Document`, instead it will be based on a specific storage implementation like `ArrayDoc` or `BTDoc`.

Window hierarchy

As is commonly the case with frameworks, most of the classes are involved with user interaction, both data display and data input. In Figure 30.4, these classes are represented by the "Window" hierarchy. (Figure 30.4 also shows class `WindowRep`. This serves much the same role as the `WindowRep` class used Chapter 29; it encapsulates the low level details of how to communicate with a cursor addressable screen.)

class Window

The basic `Window` class is an extended version of that used in Chapter 29. It possesses the same behaviours of knowing its size and location on the screen, maintaining "foreground" and "background" images, setting characters in these images etc. In addition, these `Window` objects can own "subwindows" and can arrange that these subwindows get displayed. They can also deal with display of text strings and numbers at specific locations.

NumberItem

Class `NumberItem` is a minor reworking of the version from Chapter 29. An instance of class `NumberItem` can be used to display the current value of a variable and can be updated as the variable is changed.

EditWindow

The simple `EditText` class of Chapter 29 has been replaced by an entire hierarchy. The new base class is `EditWindow`. `EditWindow` objects are things that can be told to "handle input". "Handling input" involves accepting and processing input characters until a '`\n`' character is entered.

EditNum and EditText

Class `EditNum` and `EditText` are simple specializations that can be used for verified numeric or text string input. An `EditNum` object accepts numeric characters, using them to determine the (integer) value input. An `EditNum` object can be told the range permitted for input data; normally it will verify that the input value is in this range (substituting the original value if an out of range value is input). An `EditText` object accepts printable characters and builds up a string (with a fixed maximum length).

MenuWindow

A `MenuWindow` allows a user to pick from a displayed list of menu items. A `MenuWindow` is built up by adding "menu items" (these have a string and a numeric identifier). When a MenuWindow is displayed, it shows its menu items along with an indicator of "the currently selected item" (starting at the first item in the menu). A MenuWindow handles "tab" characters (other characters are ignored). A "tab" changes the currently selected item. The selection moves cyclically through the list of items. When "enter" ('\n') is input, the MenuWindow returns the numeric identifier associated with the currently selected menu item.

Classes NumberDialog and TextDialog

The "dialogs" display small windows centered in the screen that contain a prompt and an editable field (an instance of class EditNum or class EditText). The user must enter an acceptable value before the dialog will disappear and the program continue. Class InputFileDialog is a minor specialization of TextDialog that can check whether a string given as input corresponds to the name of an existing file.

RecordWindow

Class RecordWindow is a slightly more elaborate version of class MenuWindow. A RecordWindow owns a list of EditNum and EditText subwindows. "Tabbing" in a RecordWindow selects successive subwindows for further input.

The "MyRec" class used in a particular program implements a function, AddFieldsToWindow(), that populates a RecordWindow with the necessary EditNum and EditText subwindows.

30.3 THE COMMAND HANDLER CLASSES

30.3.1 Class declarations

CommandHandler

The declaration of class CommandHandler is:

```
class CommandHandler {
public:
    CommandHandler(int mainmenuid);
    virtual ~CommandHandler();

    virtual void  Run();
protected:
    virtual void  MakeMenu() = 0;
    virtual void  CommandLoop();
    virtual void  PrepareToRun() { }
    virtual void  HandleCommand(int command) = 0;
    virtual void  UpdateState() { }
    virtual void  Finish() { }

    MenuWindow    *fMenu;
    int           fFinished;
    int           fMenuID;
};
```

A CommandHandler is basically something that owns a MenuWindow (with an associated integer identifier). A CommandHandler can "run". It does this by filling in the menu entries, "preparing to run", executing its command loop, and finally tidying up. The command loop will involve an update of status, acceptance of a command number from the MenuWindow and execution of HandleCommand(). One of the commands must set the fFinished flag.

Functions with empty bodies

Some of the functions are declared with "empty" implementations, e.g. PrepareToRun(). Such functions are fairly common in frameworks. They represent points where the framework designer has made provision for "unusual" behaviours that might be necessary in specific programs.

Usually there is nothing that must be done before an Application displays its main menu, or after its command handling loop is complete. But it is possible that a particular program would need special action (e.g. display of a "splash screen" that identifies the program). So, functions PrepareToRun() and Finish() are declared and are called from within defined code, like that of function Run(). These functions are deliberately given empty definitions (i.e. { }) ; there is no need to force every program to define actual implementations.

Pure virtual functions

In contrast functions like MakeMenu() and HandleCommand() are pure virtual. These have to be given definitions before you have a working program.

Application

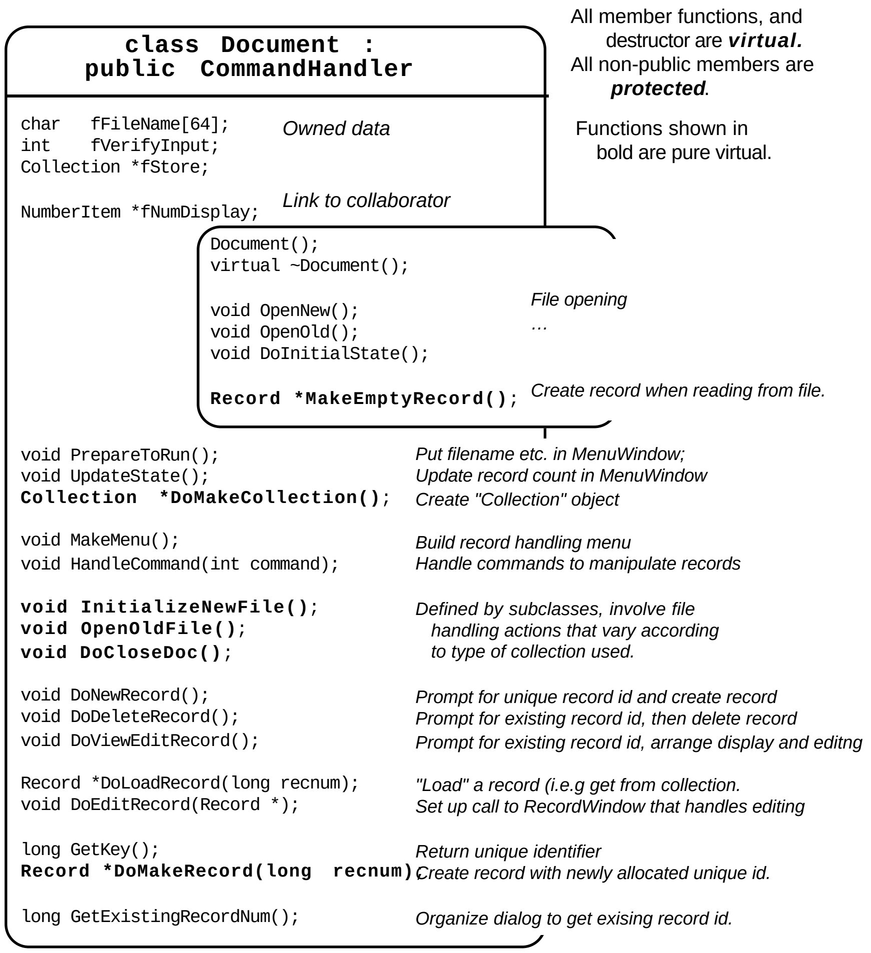
Class Application is a minor specialization of CommandHandler. An Application is a kind of CommandHandler that owns a Document that it creates in its DoMakeDocument() function. An Application provides effective implementations for the pure virtual functions CommandHandler::MakeMenu() and Command Handler::HandleCommand().

```
class Application : public CommandHandler {
public:
    Application();
    virtual ~Application();
protected:
    virtual void  MakeMenu();
    virtual void  HandleCommand(int command);
    virtual       Document* DoMakeDocument() = 0;
    Document      *fDoc;
};
```

Function DoMakeDocument() is in the protected section. In a normal program, it is only used from within HandleCommand() and consequently it does not need to be public. It is not made private because it is possible that it might need to be called from a function defined in some specialized subclass of class Application. Because function DoMakeDocument() is pure virtual, class Application is still abstract. Real programs must define a specialized subclass of class Application.

Document

Class Document is a substantially more elaborate kind of CommandHandler; Figure 30.5 is a design diagram for the class.



```
class Document :
    public CommandHandler

char   fFileName[64];
int    fVerifyInput;
Collection *fStore;

NumberItem *fNumDisplay;

            Document();
            virtual ~Document();

            void OpenNew();
            void OpenOld();
            void DoInitialState();

            Record *MakeEmptyRecord();

void PrepareToRun();
void UpdateState();
Collection  *DoMakeCollection();

void MakeMenu();
void HandleCommand(int command);

void InitializeNewFile();
void OpenOldFile();
void DoCloseDoc();

void DoNewRecord();
void DoDeleteRecord();
void DoViewEditRecord();

Record *DoLoadRecord(long recnum);
void DoEditRecord(Record *);

long GetKey();
Record *DoMakeRecord(long  recnum);

long GetExistingRecordNum();
```

All member functions, and destructor are ***virtual.***
All non-public members are ***protected***.
Functions shown in bold are pure virtual.
Owned data
Link to collaborator
File opening
...
Create record when reading from file.
Put filename etc. in MenuWindow;
Update record count in MenuWindow
Create "Collection" object
Build record handling menu
Handle commands to manipulate records
Defined by subclasses, involve file handling actions that vary according to type of collection used.
Prompt for unique record id and create record
Prompt for existing record id, then delete record
Prompt for existing record id, arrange display and editng
"Load" a record (i.e.g get from collection.
Set up call to RecordWindow that handles editing
Return unique identifier
Create record with newly allocated unique id.
Organize dialog to get exising record id.

Figure 30.5 Design diagram for class Document.

Document's data members

As shown in Figure 30.5, in addition to the data members that it has because it is a `CommandHandler` (e.g. the `MenuWindow`) a `Document` has, as owned data members, the file "base name" (a character array), an integer flag whose setting determines whether file names should be verified, and a `Collection` object separately created in the heap and accessed via the `fStore` pointer. (Of course, this will point to an instance of some specialized subclass of class `Collection`.) A `Document` also has a link a `NumberItem`; this (display window) object gets created by the `Document` but ownership is transferred to the `MenuWindow`.

In some cases, the contents of `fFileName` will be the actual file name. But in other cases, e.g. with the BTree storage structure, there will be separate index and data files and `fFileName` is used simply as a base name. If a single file is used for storage, then file names can be checked when opening an old file.

All the member functions, and the destructor, are `virtual` so as to allow redefinition as needed by subclasses. Quite a few functions are still pure virtual;

examples include the functions for creating Record objects and some of those involved in file handling. The data members, and auxiliary member functions, are all protected (rather than private). Again, this is done so as to maximize the potential for adaptation in subclasses.

The public interface defines a few additional functions; most of these involve file handling and related initialization and are used in the code of Application::HandleCommand().

```
class Document : public CommandHandler {
public:
    Document();
    virtual ~Document();

    virtual void        OpenNew();
    virtual void        OpenOld();
    virtual void        DoInitialState();
```

The other additional public function is one of two used to create records. Function MakeEmptyRecord() is used (by a Collection object) to create an empty record that can then be told to read data from a file:

```
    virtual Record      *MakeEmptyRecord() = 0;
```

The other record creating function, DoMakeRecord() is used from within Document::DoNewRecord(); it creates a record with a new identifier. Since it is only used from within class Document, it is part of the protected interface.

Redefining empty and pure virtual functions inherited from CommandHandler

```
protected:
    virtual void        PrepareToRun();
    virtual void        UpdateState();

    virtual void        MakeMenu();
    virtual void        HandleCommand(int command);
```

The default implementations of the protected functions PrepareToRun() and UpdateState() simply arrange for the initial display, and later update, of the information with the filename and the number of records.

Functions MakeMenu() and HandleCommand() define and work with the standard menu with its options for creating, viewing, and deleting records. These definitions cover for the pure virtual functions defined in the base CommandHandler class.

Dialogs for getting record identifiers

Class Document has two functions that use dialogs to get keys for records. Function GetKey() is used to get a new unique key, while GetExistingRecordNum() is used to get a key already associated with a record. Keys are restricted to positive non zero integers. Function GetKey() checks whether the given key is new by requesting the Collection object to find a record with the given identifier number. The function fails (returns -1) if the Collection object successfully finds a record with the given identifier (an "alert" can then be displayed warning the user that the identifier is already in use).

```
    virtual long        GetKey();
    virtual long        GetExistingRecordNum();
```

Default definitions of command handling functions

Class Document can provide default definitions for the functions that handle "new record", "view record" and "delete record" commands. Naturally, these commands are handled using auxiliary member functions called from HandleCommand():

```
    virtual void        DoNewRecord();
    virtual void        DoDeleteRecord();
    virtual void        DoViewEditRecord();
```

For example, DoViewEditRecord() can define the standard behaviour as involving first a prompt for the record number, handling the case of an invalid record number through an alert while a valid record number leads to calls to "load" the record and then the invocation of DoEditRecord().

Loading and editing records

```
    virtual Record      *DoLoadRecord(long recnum);
    virtual void        DoEditRecord(Record *r);
```

Function DoEditRecord() can ask the Record to create its own RecordWindow display and then arrange for this RecordWindow to handle subsequent input (until an "enter", '\n', character is used to terminate that interaction).

Pure virtual member functions

The remaining member functions are all pure virtual. The function that defines the type of Collection to use is defined by a subclass defined in the framework, e.g. ArrayDoc or BTDoc. These classes also provide effective definitions for the remaining file handling functions.

Functions provided in framework subclasses

```
    virtual Collection  *DoMakeCollection() = 0;
    virtual void        InitializeNewFile() = 0;
    virtual void        OpenOldFile() = 0;
    virtual void        DoCloseDoc() = 0;
```

Function DoMakeRecord(), like the related public function MakeEmptyRecord(), must be defined in a program specific subclass ("MyDoc" etc).

Program specific function

```
    virtual Record      *DoMakeRecord(long recnum) = 0;
```

Data members

The class declaration ends with the specification of the data members and links to collaborators:

```
    char          fFileName[64];
    NumberItem    *fNumDisplay;
    Collection    *fStore;
    int           fVerifyInput;
};
```

Classes BTDoc and ArrayDoc

Classes BTDoc and ArrayDoc are generally similar. They have to provide implementations of the various file handling and collection creation functions declared in class Document. Class BTDoc has the following declaration:

```
class BTDoc : public Document {
public:
    BTDoc();
protected:
    virtual Collection *DoMakeCollection();
    virtual void  InitializeNewFile();
    virtual void  OpenOldFile();
    virtual void  DoCloseDoc();
};
```

The constructor for class BTDoc simply invokes the inherited Document constructor and then changes the default setting of the "file name verification" flag (thus switching off verification). Since a BTree uses multiple files, the input file dialog can't easily check the name.

The declaration of class ArrayDoc is similar. Since it doesn't require any special action in its constructor, the interface consists of just the three protected functions.

30.3.2 Interactions

Principal interactions when opening a file

Figure 30.6 illustrates some of the interactions involved when an application object deals with an "Open" command from inside its HandleCommand() function. The illustration is for the case where the concrete "MyDoc" document class is derived from class BTDoc.

Inherited pattern of interactions

All the interactions shown in Figure 30.6 are already implemented in the framework code. A program built using the framework simply inherits the behaviour patterns shown

A single program specific function call

The only program specific aspect is the implementation of the highlighted call to DoMakeDocument(). This call to DoMakeDocument() is the first step in the process; it results in the creation of the specialized MyDoc object.

Create the collection

Once created, the new document object is told to perform its DoInitialState() routine in which it creates a Collection object. Since the supposed MyDoc class is derived from class BTDoc, this step results in a new BTCollection.

Open the file

The next step, opening the file, is relatively simple in the case of a BTDoc. First the document uses a dialog to get the file name; an InputFileDialog object is created temporarily for this purpose. Once the document has the file name, it proceeds by telling its BTreeCollection to "open the BTree". This step leads to the creation of the actual BTree object whose constructor opens both index and data files.

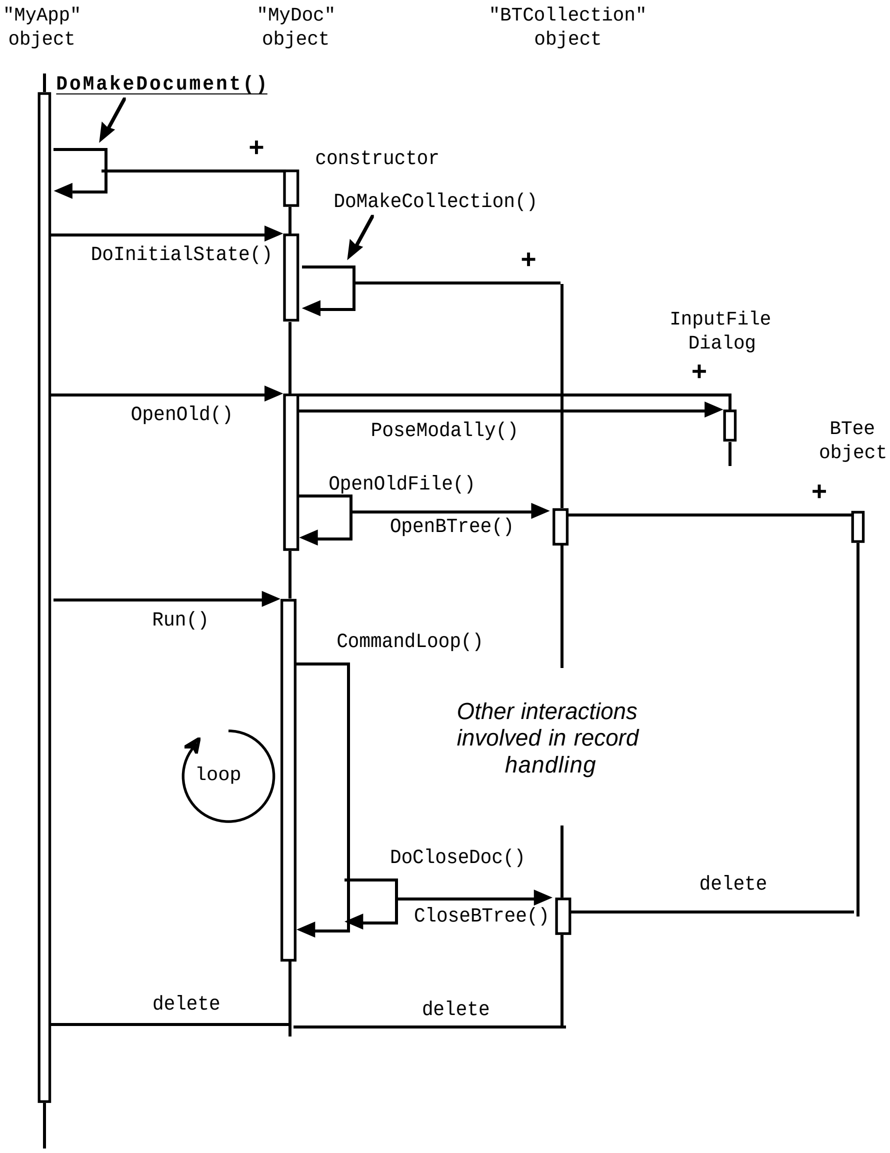


Figure 30.6 Framework defined interactions for an Application object handling an "Open" command.

The next step would get the document to execute its `Run()` function. This would involve processing any record handling commands (none shown in Figure 30.6) and eventually a "Close" command.

Closing

A "Close" command gets forwarded to the collection. This responds by deleting the `BTree` object (whose destructor arranges to save all "housekeeping data" and then close its two files).

A return is then made from Document::Run() back to Application::HandleCommand(). Since the document object is no longer needed, it gets deleted. As shown in Figure 30.6 this also leads to the deletion of the BTCollection object.

Additional interactions for a memory based collection

A document based on an "in memory" storage structure would have a slightly more elaborate pattern of interactions because it has to load the existing records when the file is opened. The overall pattern is similar. However, as shown in Figure 30.7, the OpenOldFile() function results in additional interactions.

In this case, the collection object (an ADCollection) will be told to read all its data from the file. This will involve control switching back and forth among several objects as shown in Figure 30.7. The collection object would read the number of records and then have a loop in which records get created, read their own data, and are then added to the DynamicArray. The collection object has to ask the document object to actually create the new Record (this step, and the MyRec::ReadFrom() function, are the only program specific parts of the pattern).

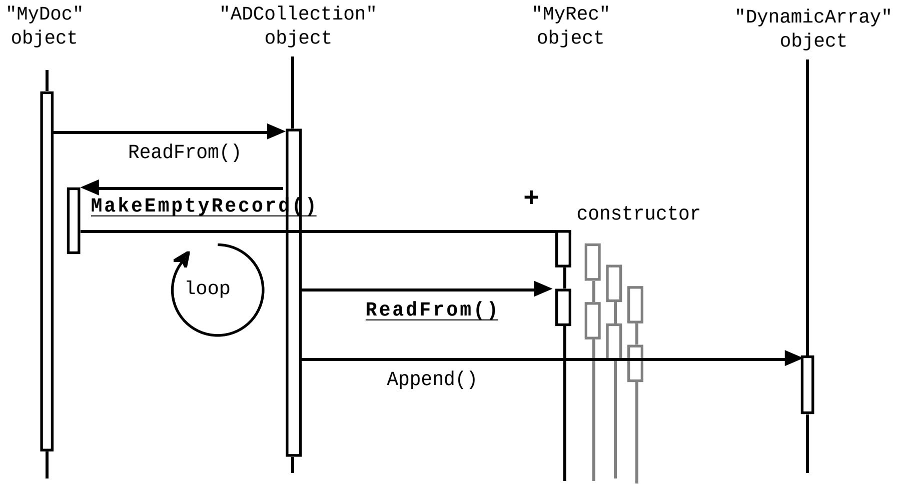


Figure 30.7 Opening a document whose contents are memory resident (interactions initiated by a call to ArrayDoc::OpenOldFile()).

Interactions while creating a new record

Figure 30.8 illustrates another pattern of interactions among class instances that is almost entirely defined within the framework code. It shows the overall steps involved in creating a new record (Document::DoNewRecord() function).

The interactions start with the document object using a NumberDialog object to get a (new) record number from the user. The check to determine that the number is new involves a request to the collection object to perform a Find() operation (this Find() should fail).

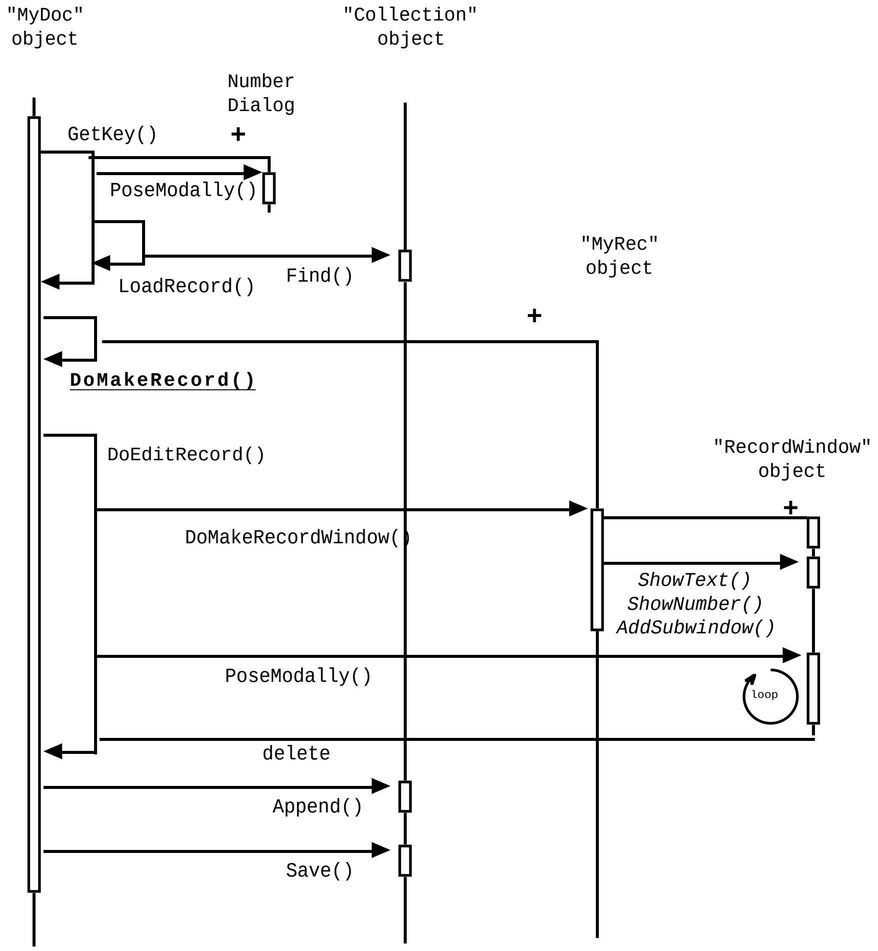


Figure 30.8 Some of the interactions resulting from a call to Document:: DoNewRecord().

If the newly entered record identifier number is unique, a new record is created with the given record number. This involves the program specific implementation of the function `MyDoc::DoMakeRecord()`.

The next step involves the new `MyRec` object being asked to create an appropriate `RecordWindow`. The object created will be a standard `RecordWindow`; but the code for `MyRec::DoMakeRecordWindow()` will involve program specific actions adding a specific set of text labels and editable fields.

Once created, the `RecordWindow` will be "posed modally". It has a loop dealing with subsequent input. In this loop it will interact with the `MyRec` object (notifying it of any changes to editable fields) as well as with the editable subwindows that it contains.

When control is returned from `RecordWindow::PoseModally()`, the `MyRec` object will already have been brought up to date with respect to any changes. The

RecordWindow can be deleted. The new record can then be added to the Collection (the Append() operation).

Although the new record was created by the document object, it now belongs to the collection. The call to Save() represents an explicit transfer of ownership. Many collections will have empty implementations for Save() (because they don't have to do anything special with respect to ownership). A collection that uses a BTree will delete the memory resident record because it will already have made a permanent copy on disk (during its Append() operation).

30.3.3 Implementation Code

CommandHandler

The constructor for class CommandHandler sets is state as "unfinished" and creates a MenuWindow. (All Window objects have integer identifiers. In the current code, these identifiers are only used by RecordWindow and Record objects.)

```
CommandHandler::CommandHandler(int mainmenuid)
{
    fMenuID = mainmenuid;
    fMenu = new MenuWindow(fMenuID);
    fFinished = 0;
}
```

The destructor naturally gets rid of the MenuWindow.

```
CommandHandler::~CommandHandler()
{
    delete fMenu;
}
```

The Run() function completes the construction of the MenuWindow and other initial preparations and then invokes the CommandLoop() function. When this returns, any special tidying up operations are performed in Finish().

```
void CommandHandler::Run()
{
    this->MakeMenu();
    this->PrepareToRun();
    this->CommandLoop();
    this->Finish();
}
```

It may appear that there is something slightly odd about the code. The MenuWindow is created in the CommandHandler's constructor, but the (virtual) function MakeMenu() (which adds menu items to the window) is not called until the start of Run(). It might seem more natural to have the call to MakeMenu() as part of the CommandHandler's constructor.

However, that arrangement does not work. *Constructors do not use dynamic calls to virtual functions.* If the call to MakeMenu() was part of the CommandHandler's constructor, it would be the (non existent) CommandHandler::MakeMenu() function that was invoked and not the desired Application::MakeMenu() or Document::MakeMenu() function.

Beware: no "virtual constructors"

Constructors that do invoke virtual functions dynamically (so called "virtual constructors") are a frequently requested extension to C++ but, for technical reasons, they can not be implemented.

Class CommandHandler provides the default implementation for one other function, the main CommandLoop():

```
void CommandHandler::CommandLoop()
{
    while(!fFinished) {
           this->UpdateState();
           int c = fMenu->PoseModally();
           this->HandleCommand(c);
           }
}
```

Application

Class Application is straightforward. Its constructor and destructor add nothing to those defined by the base CommandHandler class:

```
Application::Application() : CommandHandler(kAPPMENU_ID)
{
}

Application::~Application()
{
}
```

Its MakeMenu() function adds the three standard menu items to the MenuWindow. (Constants like kAPPEMENU_ID, cNEW and related constants used by class Document are all defined in a common header file. Common naming conventions have constants that represent "commands" take names starting with 'c' while those that define other parameters have names that begin with 'k'.)

```
void Application::MakeMenu()
{
    fMenu->AddMenuItem("New", cNEW);
    fMenu->AddMenuItem("Open",cOPEN);
    fMenu->AddMenuItem("Quit",cQUIT);
}
```

The code implementing Application::HandleCommand() was given earlier (in Section 30.2).

Document

The constructor for class Document has a few extra data members to initialize. Its destructor gets rid of the fStore (Collection object) if one exists.

```
Document::Document() : CommandHandler(kDOCMENU_ID)
{
    fStore = NULL;
    fFileName[0] = '\0';
    fVerifyInput = 1;
}
```

The MakeMenu() function adds the standard menu options to the document's MenuWindow() function. Member DoInitialState() simply uses the (pure virtual) DoMakeCollection() function to make an appropriate Collection object. The function actually called would be BTDoc::DoMakeCollection() or ArrayDoc::DoMakeCollection (or other similar function) depending on the particular type of document that is being built.

Initialization functions

```
void Document::MakeMenu()
{
    fMenu->AddMenuItem("New record", cNEWREC);
    fMenu->AddMenuItem("Delete record",cDELREC);
    fMenu->AddMenuItem("View/edit record",cVIEW);
    fMenu->AddMenuItem("Close file",cCLOSE);
}

void Document::DoInitialState()
{
    fStore = DoMakeCollection();
}
```

Display of document details

Functions PrepareToRun() and UpdateState() deal with the initial display and subsequent update of the display fields with document details. (The NumberItem used for output is given to the MenuWindow as a "subwindow". As explained in Section 30.6, subwindows "belong" to windows and, when appropriate, get deleted by the window. So although the Document creates the NumberItem and keeps a link to it, it never deletes it.)

```
void Document::PrepareToRun()
{
    fNumDisplay = new NumberItem(0, 31, 1, 40,
            "Number of records:",0);
    fMenu->AddSubWindow(fNumDisplay);
}

void Document::UpdateState()
{
    fNumDisplay->SetVal(fStore->Size());
    fMenu->ShowText(fFileName, 2, 2, 30, 0, 1);
}
```

Getting keys for new and existing records

The functions `GetKey()` and `GetExistingRecordNum()` both use dialogs for input. Function `GetKey()`, via the `DoLoadRecord()` function, lets the document interact with the `Collection` object to verify that the key is not already in use:

```
long Document::GetKey()
{
    NumberDialog n("Record identifier", 1, LONG_MAX);
    long k = n.PoseModally(1);
    if(NULL == DoLoadRecord(k))
           return k;
    Alert("Key already used");
    return -1;
}

long Document::GetExistingRecordNum()
{
    if(fStore->Size() == 0) {
           Alert("No records defined.");
           return -1;
           }
    NumberDialog n("Record number", 1, LONG_MAX);
    return n.PoseModally(1);
}

Record *Document::DoLoadRecord(long recnum)
{
    return fStore->Find(recnum);
}
```

Provision for further extension

Function `DoLoadRecord()` might seem redundant, after all the request to `fStore` to do a `Find()` operation could have been coded at the point of call. Again, this is just a point where provision for modification has been built into the framework. For example, someone working with a disk based collection of records might want to implement a version that maintained a small "cache" or records in memory. He or she would find it convenient to have a redefinable `DoLoadRecord()` function because this would represent a point where the "caching" code could be added.

Function `Alert()` is defined along with the windows code. It puts up a dialog displaying an error string and waits for the user to press the "enter" key.

Running and handling commands

Class `Document` uses the inherited `CommandHandler::Run()` function. Its own `HandleCommand()` function was given earlier (Section 30.2). `Document::HandleCommand()` is implemented in terms of the auxiliary member functions `DoNewRecord()`, `DoDeleteRecord()`, `DoViewEditRecord()`, and `DoCloseDoc()`. Function `DoCloseDoc()` is pure virtual but the others have default definitions.

Functions `DoNewRecord()` and `DoEditRecord()` implement most of the interactions shown in Figure 30.8; getting the key, making the record, arranging for it to be edited through a temporary `RecordWindow` object, adding and transferring ownership of the record to the `Collection` object:

```
void Document::DoNewRecord()
{
    long key = GetKey();
    if(key<=0)
            return;
    Record *r = DoMakeRecord(key);
    DoEditRecord(r);
    fStore->Append(r);
    fStore->Save(r);
}

void Document::DoEditRecord(Record *r)
{
    RecordWindow *rw = r->DoMakeRecordWindow();
    rw->PoseModally();
    delete rw;
}
```

Function `DoDeleteRecord()` gets a record identifier and asks the `Collection` object to delete that record. (The `Collection` object is expected to return a success indicator. A failure results in a warning to the user that the key given did not exist.)

```
void Document::DoDeleteRecord()
{
    long recnum = GetExistingRecordNum();
    if(recnum<0)
            return;
    if(!fStore->Delete(recnum))
            Alert(NoRecMsg);
}
```

Function `DoViewEditRecord()` first uses a dialog to get a record number, and then "loads" that record. If the load operation fails, the user is warned that the record number was invalid. If the record was loaded, it can be edited:

```
void Document::DoViewEditRecord()
{
    long recnum = GetExistingRecordNum();
    if(recnum<0)
            return;
    Record *r = DoLoadRecord(recnum);
    if(r == NULL) {
            Alert(NoRecMsg);
            return;
            }
    DoEditRecord(r);
    fStore->Save(r);
}
```

Standard file handling

Functions `OpenNew()` and `OpenOld()` handle standard aspects of file opening (such as display of dialogs that allow input of a file name). Aspects that depend on the type of collection structure used are handled through the auxiliary (pure virtual) functions `InitializeNewFile()` and `OpenOldFile()`:

```
void Document::OpenNew()
{
    TextDialog onew("Name for new file");
```

```
        onew.PoseModally("example",fFileName);
        InitializeNewFile();
    }

    void Document::OpenOld()
    {
        InputFileDialog oold;
        oold.PoseModally("example",fFileName, fVerifyInput);
        OpenOldFile();
    }
```

ArrayDoc

Class `ArrayDoc` has to provide implementations for the `DoMakeCollection()` and the related file manipulation functions.

It creates an `ADCollection` (a `DynamicArray` within an "adapter"):

Creating a collection object

```
    Collection *ArrayDoc::DoMakeCollection()
    {
        return new ADCollection(this);
    }
```

The `OpenOldFile()` function has to actually open the file for input, then it must get the `ADCollection` to load all the data. Note the typecast on `fStore`. The type of `fStore` is `Collection*`. We know that it actually points to an `ADCollection` object. So we can use the typecast to get an `ADCollection*` pointer. Then it is possible to invoke the `ADCollection::ReadFrom()` function:

Handling file transfers

```
    void ArrayDoc::OpenOldFile()
    {
        fstream in;
        in.open(fFileName, ios::in | ios::nocreate);
        if(!in.good()) {
                Alert("Bad file");
                exit(1);
                }
        ((ADCollection*)fStore)->ReadFrom(in);
        in.close();
    }
```

Function `DoCloseDoc()` is very similar. It opens the file for output and then arranges for the `ADCollection` object to save the data (after this, the collection has to be told to delete all its contents):

```
    void ArrayDoc::DoCloseDoc()
    {
        fstream out;
        out.open(fFileName, ios::out);
        if(!out.good()) {
                Alert("Can not open output");
                return;
                }
        ((ADCollection*)fStore)->WriteTo(out);
        out.close();

        ((ADCollection*)fStore)->DeleteContents();

    }
```

BTDoc

As noted earlier, a `BTDoc` needs to change the default setting of the "verify file names" flag (normally set to true in the `Document` constructor):

```
    BTDoc::BTDoc() { fVerifyInput = 0; }
```

Its `DoMakeCollection()` function naturally makes a `BTCollection` object (the `this` argument provides the collection object with the necessary link back to its document):

```
    Collection *BTDoc::DoMakeCollection()
    {
        return new BTCollection(this);
    }
```

The other member functions defined for class `BTDoc` are simply an interface to functions provided by the `BTCollection` object:

Let the BTree object handle the files

```
    void BTDoc::InitializeNewFile()
    {
        ((BTCollection*)fStore)->OpenBTree(fFileName);
    }

    void BTDoc::OpenOldFile()
    {
        ((BTCollection*)fStore)->OpenBTree(fFileName);
    }

    void BTDoc::DoCloseDoc()
    {
        ((BTCollection*)fStore)->CloseBTree();
    }
```

30.4 COLLECTION CLASSES AND THEIR ADAPTERS

The underlying collection classes are identical to the versions presented in earlier chapters. Class `DynamicArray`, as used in `ADCollection`, is as defined in Chapter 21. (The `Window` classes also use instances of class `DynamicArray`.) Class `BTCollection` uses an instance of class `BTree` as defined in Chapter 24.

Class `Collection` itself is purely an interface class with the following declaration:

Abstract base class collection

```
class Collection {
public:
    Collection(Document *d) { this->fDoc = d; }
    virtual ~Collection() { }

    virtual void        Append(Record *r) = 0;
    virtual int         Delete(long recnum) = 0;
    virtual Record      *Find(long recnum) = 0;
    virtual void        Save(Record *r) { }

    virtual long        Size() = 0;
protected:
    Document            *fDoc;
};
```

It should get defined in the same file as class `Document`. A `Collection` is responsible for getting `Record` objects from disk. It has to ask the `Document` object to create a `Record` of the appropriate type; it is for this reason that a `Collection` maintains a link back to the `Document` to which it belongs.

The declarations for the two specialized subclasses are:

Specialization using an array

```
class ADCollection : public Collection {
public:
    ADCollection(ArrayDoc *d) : Collection(d) { }

    virtual void        Append(Record *r);
    virtual int         Delete(long recnum);
    virtual Record      *Find(long recnum);
    virtual long        Size();

    virtual void        ReadFrom(fstream& fs);
    virtual void        WriteTo(fstream& fs);
    virtual void        DeleteContents();

protected:
    DynamicArray  fD;
};
```

and

Specialization using a BTree

```
class BTCollection : public Collection {
public:
    BTCollection(BTDoc *d) : Collection(d) { }

    virtual void        Append(Record *r);
    virtual int         Delete(long recnum);
    virtual Record      *Find(long recnum);

    virtual long        Size();
    virtual void        Save(Record *r);

    void                OpenBTree(char* filename);
    void                CloseBTree();
private:
    BTree   *fBTree;
};
```

Each class defines implementations for the basic `Collection` functions like `Append()`, `Find()`, `Size()`. In addition, each class defines a few functions that relate to the specific form of storage structure used.

In the case of `Append()` and `Size()`, these collection class adapters can simply pass the request on to the actual collection used:

```
void ADCollection::Append(Record *r) { fD.Append(r); }

void BTCollection::Append(Record *r)
{
    fBTree->Add(*r);
}

long ADCollection::Size()
{
    return fD.Length();
}

long BTCollection::Size()
{
    return fBTree->NumItems();
}
```

Adapting a dynamic array to fulfil the role of Collection

A `DynamicArray` doesn't itself support record identifiers, so the `Find()` operation on an `ADCollection` must involve an inefficient linear search:

```
Record *ADCollection::Find(long recnum)
{
    int n = fD.Length();
    for(int i = 1; i<=n; i++) {
           Record *r = (Record*) fD.Nth(i);
           long k = r->Key();
           if(k == recnum)
                  return r;
           }
    return NULL;
}
```

This `Find()` function has to be used to implement `Delete()` because the record must first be found before the `DynamicArray::Remove()` operation can be used. (Function `Delete()` takes a record identifier, `Remove()` requires a pointer).

```
int ADCollection::Delete(long recnum)
{
    Record *r = Find(recnum);
    if(r != NULL) {
           fD.Remove(r);
           return 1;
           }
    else return 0;
}
```

The linear searches involved in most operations on an ADCollection make this storage structure unsuitable for anything apart from very small collections. A more efficient "in-memory" structure could be built using a binary tree or an AVL tree.

Adapting a BTree to fulfil the role of Collection

The corresponding functions for the BTCollection also involve work in addition to the basic Find() and Remove() operations on the collection. Function BTCollection::Find() has to return a pointer to an in-memory record. All the records in the BTree itself are on disk. Consequently, Find() had better create the in-memory record. This gets filled with data from the disk record (if it exists). If the required record is not present (the BTree::Find() operation fails) then the newly created record should be deleted.

```
Record *BTCollection::Find(long recnum)
{
    Record *r = fDoc->MakeEmptyRecord();
    int success = fBTree->Find(recnum, *r);
    if(success)
           return r;

    delete r;
    return NULL;
}
```

(The record is created via a request back to the document object: fDoc->MakeEmpty Record();.)

The function BTree::Remove() does not return any success or failure indicator; "deletion" of a non-existent key fails silently. A Collection is supposed to report success or failure. Consequently, the BTCollection has to do a Find() operation on the BTree prior to a Remove(). If this initial BTree::Find() fails, the BTCollection can report a failure in its delete operation.

```
int BTCollection::Delete(long recnum)
{
    Record *r = Find(recnum);
    if(r != NULL) {
           delete r;
           fBTree->Remove(recnum);
           return 1;
           }
    else return 0;
}
```

The other member functions for these classes mostly relate to file handling. Function ADCollection::ReadFrom() implements the scheme shown in Figure 30.7 for loading the entire contents of a collection into memory:

```
void ADCollection::ReadFrom(fstream& fs)
{
    long len;
    fs.read((char*)&len, sizeof(len));
    for(int i = 1; i <= len; i++) {
           Record *r = fDoc->MakeEmptyRecord();
           r->ReadFrom(fs);
           fD.Append(r);
           }
}
```

The `WriteTo()` function handles the output case, getting called when a document is closed. It writes the number of records, then loops getting each record to write its own data:

```
void ADCollection::WriteTo(fstream& fs)
{
    long len = fD.Length();
    fs.write((char*)&len, sizeof(len));
    for(int i = 1; i <= len; i++) {
           Record *r = (Record*) fD.Nth(i);
           r->WriteTo(fs);
           }
}
```

A `DynamicArray` does not delete its contents. (It can't really. It only has `void*` pointers). When a document is finished with, all in memory structures should be freed. The `ADCollection` has to arrange this by explicitly removing the records from the `DynamicArray` and deleting them.

```
void ADCollection::DeleteContents()
{
    int len = fD.Length();
    for(int i = len; i>= 1; i--) {
           Record* r = (Record*) fD.Remove(i);
           delete r;
           }
}
```

(The code given in Chapter 21 for class `DynamicArray` should be extended to include a destructor that does get rid of the associated array of `void*` pointers.)

The remaining functions of class `BTCollection` are all simple. Function `OpenBTree()` creates the `BTree` object itself (letting it open the files) while `CloseBTree()` deletes the `BTree`.

```
void BTCollection::OpenBTree(char* filename)
{
    fBTree = new BTree(filename);
}

void BTCollection::CloseBTree()
{
    delete fBTree;
}
```

Class `BTCollection` provides an implementation for `Save()`. This function is called whenever the `Document` object has finished with a `Record` object. In the case of an in-memory collection, like ADCollection, no action is required. But with the BTCollection, it is necessary to get rid of the now redundant in-memory record.

```
void BTCollection::Save(Record *r)
{
    delete r;
}
```

30.5 CLASS RECORD

As noted earlier, class KeyedStorableItem is simply an interface (it is the interface for storable records as defined for class BTree in Chapter 24):

```
class KeyedStorableItem {
public:
    virtual       ~KeyedStorableItem() { }
    virtual long  Key(void) const = 0;
    virtual void  PrintOn(ostream& out) const  { }
    virtual long  DiskSize(void) const = 0;
    virtual void  ReadFrom(fstream& in) = 0;
    virtual void  WriteTo(fstream& out) const = 0;
};
```

Class Record provides a default implementation for the Key() function and adds the responsibilities related to working with a RecordWindow that allows editing of the contents of data members (as defined in concrete subclasses).

```
class Record  : public KeyedStorableItem {
public:
    Record(long recNum) { this->fRecNum = recNum; }
    virtual ~Record() { }
    virtual long  Key() const { return this->fRecNum; }

    virtual RecordWindow *DoMakeRecordWindow();
    virtual void  SetDisplayField(EditWindow *e);
    virtual void  ReadDisplayField(EditWindow *e);
    virtual void  ConsistencyUpdate(EditWindow *e) { }
protected:
    virtual void  CreateWindow();
    virtual void  AddFieldsToWindow();
    long          fRecNum;
    RecordWindow  *fRW;
};
```

Data members defined by class Record

Class Record defines two data members itself. One is a long integer to hold the unique identifier (or "key"), the other is a link to the RecordWindow collaborator.

Function DoMakeRecordWindow() uses an auxiliary function CreateWindow() to actually create the window. Once created, the window is "populated" by adding subwindows (in AddFieldsToWindow()).

```
RecordWindow *Record::DoMakeRecordWindow()
{
    CreateWindow();
    AddFieldsToWindow();
    return fRW;
}
```

Function CreateWindow() is another "unnecessary" function introduced to increase the flexibility of the framework. It is unlikely that any specific program would need to change the way windows get created, but it is possible that some program might need to use a specialized subclass of RecordWindow. Having the window creation step handled by a separate virtual function makes adaptation easier.

```
void Record::CreateWindow()
{
    fRW = new RecordWindow(this);
}
```

Function Record::AddFieldsToWindow() can add any "background" text labels to the RecordWindow. A concrete "MyRec" class would also add EditNum and EditText subwindows for each editable field; an example is given later in Section 30.8. Each of these subwindows is specified by position, a label (possibly empty) and a unique identifier. These window identifiers are used in subsequent communications between the RecordWindow and Record objects.

Building the display structure

```
void Record::AddFieldsToWindow()
{
    fRW->ShowText("Record identifier ", 2, 2, 20, 0);
    fRW->ShowNumber(fRecNum, 24, 2, 10);
    // Then add data fields

    // typical code
    // EditText *et = new EditText(1001, 5, 4, 60,
    //      "Student Name ");
    //fRW->AddSubWindow(et);
    //EditNum *en = new EditNum(1002, 5, 8, 30,
    //      "Assignment 1 (5) ", 0, 5,1);
    //fRW->AddSubWindow(en);
}
```

When a RecordWindow is first displayed, it will ask the corresponding Record to set values in each of its editable subwindows. This is done by calling Record::SetDisplayField(). The EditWindow passed via the pointer parameter can be asked its identity so allowing the Record object to select the appropriate data to be used for initialization:

Showing current data values

```
void Record::SetDisplayField(EditWindow *e)
{
//  Typical code:
//  long id = e->Id();
//  switch(id) {
```

```
//case 1001:
//         ((EditText*)e)->SetVal(fStudentName, 1);
//         break;
//case 1002:
//         ((EditNum*)e)->SetVal(fMark1,1);
//         break;
//  …
}
```

Function `Record::ReadDisplayField()` is called when the contents of an `EditWindow` have been changed. The `Record` can identify which window was changed and hence determine which data member to update:

Getting newly edited values

```
void Record::ReadDisplayField(EditWindow *e)
{
//  Typical code:
//  long id = e->Id();
//  switch(id) {
//case 1001:
//  char* ptr = ((EditText*)e)->GetVal();
//  strcpy(fStudentName, ptr);
//  break;
//case 1002:
//  fMark1 = ((EditNum*)e)->GetVal();
//  break;
//  …
}
```

Interactions between `Record` and `RecordWindow` objects are considered in more detail in the next section (see Figure 30.9).

30.6 THE WINDOWS CLASS HIERARCHY

30.6.1 Class Responsibilities

WindowRep

The role of the unique `WindowRep` object is unchanged from that in the version given in Chapter 29. It "owns" the screen and deals with the low level details involved in input and output of characters.

The version used for the "RecordFile" framework has two small extensions. There is an extra public function, `Beep()`; this function (used by some dialogs) produces an audible tone to indicate an error. (The simplest implementation involves outputting '`\a`' "bell" characters.)

The other extension arranges for any characters input via `GetChar()` to be copied into the `fImage` array. This improves the consistency of screen updates that occur after data input.

The implementation of this extension requires additional `fXC, fYC` integer data members in class `WindowRep`. These hold the current position of the cursor. They are updated by functions like MoveCursor() and PutCharacter(). Function GetChar() stores an input character at the point (fXC, fYC) in the fImage array.

These extensions are trivial and the code is not given.

Window

Class Window is an extended version of that given in Chapter 29. The major extension is that a Window can have a list (dynamic array) of "subwindows" and so has some associated functionality. In addition, Window objects have integer identifiers and there are a couple of extra member functions for things like positioning the cursor and outputting a string starting at a given point in the Window.

The additional (protected) data members are:

```
DynamicArray      fSubWindows;  "List of subwindows"
int               fId;          "Identifier"
```

The following functions are added to the public interface:

Window identifier

```
int        Id() const { return this->fId; }
```

Function Id() returns the Window object's identifier. The constructor for class Window is changed so that this integer identifier number is an extra (first) parameter. (The destructor is extended to get rid of subwindows.)

Utility output functions

```
void       ShowText(const char* str, int x, int y,
               int width, int multiline =0, int bkgd = 1);
void       ShowNumber(long value, int x, int y, int width,
               int bkgd = 1);
void       ClearArea(int x0, int y0, int x1, int y1,
               int bkgd);
```

These functions are just generally useful output functions. Function ShowText() outputs a string, possibly on multiple lines, starting at the defined x, y position . Function ShowNumber() converts a number to a text string and uses ShowText() to output this at the given position. Function ClearArea() fills the specified area with ' 'characters. Each of these functions can operate on either the "foreground" or "background" image array of the Window. The code for these functions is straightforward and is not given.

```
virtual void SetPromptPos(int x, int y);
```

The default implementation of SetPromptPos() involves a call to the WindowRep object's MoveCursor() function. It is necessary to change the x, y values to switch from Window coordinates to screen coordinates before invoking MoveCursor().

The function PrepareToDisplay() is just an extra "hook" added to increase flexibility in the framework. It gets called just before a Window gets displayed allowing for any unusual special case initializations to be performed.

```
virtual void PrepareToDisplay() { }
```

Provision for subwindows

The first of the extra functions needed to support subwindows is CanHandleInput(). Essentially this returns "true" if a Window object is actually an instance of some class derived from class EditWindow where a GetInput() function is defined. In some situations, it is necessary to know which (sub)windows are editable so this function has been added to the base class for the entire windows hierarchy. By default, the function returns "false".

```
virtual    int    CanHandleInput() { return 0; }
```

The three main additions for subwindows are the public functions AddSubWindow() and DisplayWindow() and the protected function Offset().

```
void       AddSubWindow(Window *w);
void       Offset(int x, int y);
void       DisplayWindow();
```

Member function AddSubWindow(Window *w) adds the given Window w as a subwindow of the Window executing the function. This just involves appending w to the "list" fSubWindows.

When creating a subwindow, it is convenient to specify its position in terms of the enclosing window. However, the fX, fY position fields of a Window are supposed to be in screen coordinates. The function Offset(), called from AddSubWindow(), changes the fX, fY coordinates of a subwindow to take into account the position of its enclosing parent window.

The function DisplayWindow(), whose implementation is given later, prepares a window, and its subwindows for display. This involves calls to functions such as PrepareContent(), PrepareToDisplay(), and ShowAll().

NumberItem

The role of class NumberItem is unchanged from that of the version presented in the last chapter; it just displays a numeric value along with a (possibly null) label string.

The constructor now has an additional "int id" argument at the front of the argument list that is used to set a window identifier. Function SetVal() also has an extra "int redraw" parameter; if this is false (which is the default) a change to the value does not lead to immediate updating of the screen.

The implementation of NumberItem was changed to use the new functionality like ShowText() and ShowNumber() added to the base Window class. The implementation code is not given, being left as an exercise.

EditWindow

The significant changes to the previously illustrated Window classes start with class EditWindow.

Class EditWindow is intended to be simply an abstraction. It represents a Window object that can be asked to "get input".

Getting input

"Getting input" means different things in different contexts. When an EditNum window is "getting input" it consumes digits to build up a number. A MenuWindow "gets input" by using "tab" characters to change the selected option. However, although they differ in detail, the various approaches to "getting input" share a similar overall pattern.

Overall input process

There may have to be some initialization. After this is completed, the actual input step can be performed. Sometimes, it is necessary to validate an input. If an input value is unacceptable the user should be notified and then the complete process should be repeated.

Consuming characters

The actual input step itself will involve a loop in which characters are accepted (via the WindowRep object) and are then processed. Different kinds of EditWindow process different kinds of character; thus a MenuWindow can basically ignore everything except tabs, an EditNum only wants to handle digits, while an EditText can handle more or less any (printable) character. The character input step should continue until some terminator character is entered. (The terminator character may be subclass specific.) The terminator character is itself sometimes significant; it should get returned as the result of the GetInput() function.

This overall pattern can be defined in terms of a GetInput() function that works with auxiliary member functions that can be redefined in subclasses. Pseudo-code for this GetInput() function is as follows:

Inner loop getting characters until terminator

Validate

```
InitializeInput();
do {
    SetCursorPosition();
    Get character (ch)
    while(ch != '\n') {
            if(!HandleCharacter(ch))
                    break;
            Get character (ch)
            }
    TerminateInput();
    v = Validate();
} while (fMustValidate && !v);
return ch;
```

The outer loop, the do … while() loop, makes the EditWindow keep on processing input until a "valid" entry has been obtained. (An entry is "valid" if it satisfies the Validate() member function, or if the fMustValidate flag is false).

Prior to the first character input step, the cursor is positioned, so that input characters appear at a suitable point within the window.

The "enter" key ('\n') is to terminate all input operations. The function HandleCharacter() may return false if the character ch corresponds to some other (subclass specific) terminator; if this function returns true it means that the character was "successfully processed" (possibly by being discarded).

The virtual function `TerminateInput()` gets called when a terminating character has been read. Class `EditText` is an example of where this function can be useful; `EditText::TerminateInput()` adds a null character ('`\0`') at the end of the input string.

In addition to checking the data entered, the `Validate()` function should deal with aspects like notifying a user of an incorrect value.

Class `EditWindow` can provide default definitions for most of these functions. These defaults make all characters acceptable (but nothing gets done with input characters), make all inputs "valid", do nothing special at input termination etc. Most subclasses will need to redefine several if not all these virtual functions.

By default, an `EditWindow` is expected to be a framed window with one line of content area (as illustrated in Figure 30.1 with its editable name and number fields).

The class declaration is:

EditWindow declaration

Main GetInput() function

Auxiliary functions for GetInput()

```
class EditWindow : public Window {
public:
    EditWindow(int id, int x, int y, int width,
                  int height = 3, int mustValidate = 0);
    virtual char  GetInput();

    virtual int   CanHandleInput() { return 1; }
    virtual int   ContentChanged() { return 0; }
protected:
    virtual void  InitializeInput() {
           this->PrepareToDisplay();
           }
    virtual void  SetCursorPosition() {
           this->SetPromptPos(fWidth-1, fHeight-1);
           }
    virtual int   HandleCharacter(char ch) { return 1; }
    virtual void  TerminateInput() { }
    virtual int   Validate() { return 1; }
    int           fEntry;
    int           fMustValidate;
};
```

Other member functions

The inherited function `Window::CanHandleInput()` is redefined. As specified by `EditWindow::CanHandleInput()`, all `EditWindow` objects can handle input.

A `ContentChanged()` function is generally useful For example, an `EditNum` object has an "initial value" which it displays prior to an input operation; it can detect whether the input changes this initial value. If an `EditNum` object indicates that it has not been changed, there is no need to copy its value into a `Record` object. The `ContentChanged()` behaviour might as well be defined for class `EditWindow` although only it is only useful for some subclasses. It can be given a default definition stating "no change".

Functions like `ContentChanged()` and `InitializeInput()` have been defined in the class declaration. This is simply for convenience in presentation. They should either be defined separately in the header file as `inline` functions, or defined normally in the implementation file. The definition of GetInput(), essentially the same as the pseudo code given, is included in Section 30.6.2.

EditWindow's extra data members

The two data members added by class EditWindow are a flag to indicate whether input values should be validated (fMustValidate) and the variable fEntry (which is used in several subclasses to do things like record how many characters have been accepted).

EditText

An EditText is an EditWindow that can accept input of a string. This string is held (in a 256 character buffer) within the EditText object. Some other object that needs text input can employ an EditText, require it to perform a GetInput() operation, and, when input is complete, can ask to read the EditText object's character buffer.

Naturally, class EditText must redefine a few of those virtual functions declared by class EditWindow. An EditWindow can simply discard characters that are input, but an EditText must save printable characters in its text buffer; consequently, HandleCharacter() must be redefined. An EditWindow positions the cursor in the bottom right corner of the window (an essentially arbitrary position), an EditText should locate the cursor at the left hand end of the text input field; so function SetCursorPosition() gets redefined.

Normally, there are no "validation" checks on text input, so the default "do nothing" functions like EditWindow::Validate() do not need to be redefined.

The declaration for class EditText is:

```
class EditText: public EditWindow {
public:
    EditText(int id, int x, int y, int width, char *label,
           short size = 256, int mustValidate = 0);
    void          SetVal(char* val, int redraw = 0);
    char          *GetVal() { return this->fBuf; }

    virtual int   ContentChanged();
protected:
    virtual void  InitializeInput();
    virtual void  SetCursorPosition();
    virtual int   HandleCharacter(char ch);
    virtual void  TerminateInput();

    void          ShowValue(int redraw);

    int           fLabelWidth;
    char          fBuf[256];
    int           fSize;
    int           fChanged;
};
```

Extra functions to set, and read string

Redefining auxiliary functions of GetInput()

Extra data members

Class EditText adds extra public member functions SetVal() and GetVal() that allow setting of the initial value, and reading of the updated value in its fBuf text buffer. (There is an extra protected function ShowValue() that gets used in the implementation of SetVal().)

Data members

Class EditText adds several data members. In addition to the text buffer, fBuf, there is an integer fLabelWidth that records how much of the window's width is taken up by a label. The fSize parameter has the same role as the size parameter in the EditText class used in Chapter 29. It is possible to specify a maximum size for the string. The input process will terminate when this number of characters has been entered. The class uses an integer flag, fChanged, that gets set if any input characters get stored in fBuf (so changing its previous value).

EditNum

An EditNum is an EditWindow that can deal with the input of a integer. This integer ends up being held within the EditNum object. Some other object that needs integer input can employ an EditNum, require it to perform a GetInput() operation, and, when input is complete, can ask the EditNum object for its current value.

The extensions for class EditNum are similar to those for class EditText. The class adds functions to get and set its integer. It provides effective implementations for HandleCharacter() and SetCursorPosition(). It has an extra (protected) ShowValue() function used by its SetVal() public function.

Validating numeric input

Class EditNum() redefines the Validate() function. The constructor for class EditNum requires minimum and maximum allowed values. If the "must validate" flag is set, any number entered should be checked against these limits.

The class declaration is:

Extra functions to set, and read integer

Redefining auxiliary functions of GetInput()

Data members

```
class EditNum: public EditWindow {
public:
    EditNum(int id, int x, int y, int width, char *label,
           long min, long max, int mustValidate = 0);
    void          SetVal(long, int redraw = 0);
    long          GetVal() { return this->fVal; }

    virtual int   ContentChanged();

protected:
    virtual void  InitializeInput();
    virtual void  SetCursorPosition();
    virtual int   HandleCharacter(char ch);
    virtual void  TerminateInput();
    virtual int   Validate();

    void          ShowValue(int redraw);

    int           fLabelWidth;
    long          fMin;
    long          fMax;
    long          fSetVal;
    long          fVal;
    int           fsign;
};
```

The data members include the minimum and maximum limits, the value of the EditNum, its "set value", and a label width. The fsign field is used during input to note the ± sign of the number.

MenuWindow

A MenuWindow is a specialized input handling (EditWindow) type of window that allows selection from a menu. By default, it is a "full screen" window, (70x20).

The class declares two extra public member functions: AddMenuItem() and PoseModally().

AddMenuItem()

Function AddMenuItem() adds a menu item (a string and integer combination) to the MenuWindow. There is a limit on the number of items, function AddMenuItem() returns "false" if this limit is exceeded. The limit is defined by the constant kMAXCHOICES (a suitable limit value is 6). The menu strings are written into the window's "background image" at fixed positions; their identifier numbers are held in the array fCmds.

PoseModally() and "modal" windows

Function PoseModally() does some minor setting up, calls GetInput() (using the standard version inherited from class EditWindow) and finally returns the selected command number. (A "modal" window is one that holds the user in a particular processing "mode". When a MenuWindow is "posed modally", the only thing that a user can do is switch back and forth among the different menu choices offered.)

Two of the auxiliary functions for GetInput() have to be redefined. Function SetCursorPosition() now displays a cursor marker (the characters ==>). The position is determined by the "currently chosen" menu item (as identified by the data member fChosen).

Function HandleCharacter() is redefined. This version ignores all characters apart from "tabs". Input of a "tab" character changes the value of fChosen (which runs cyclically from 0…fChoices-1 where fChoices is the number of menu items added). Whenever the value of fChosen is changed, the cursor is repositioned. (The extra private member function ClearCursorPosition() is needed to clear the previous image of the cursor.)

MenuWindow declaration

```
class MenuWindow : public EditWindow {
public:
    MenuWindow(int id, int x = 1, int y = 1,
                int width = 70, int height = 20);
    int         AddMenuItem(const char *txt, int num);
    int         PoseModally();
protected:
    virtual void  SetCursorPosition();
    virtual int   HandleCharacter(char ch);

    void        ClearCursorPosition();

    int         fCmds[kMAXCHOICES];
    int         fChoices;
    int         fChosen;
```

Additional functionality

Redefining auxiliary functions for GetInput()

Data members

```
};
```

The Dialogs: NumberDialog, TextDialog, and InputFileDialog

The "dialogs" (NumberDialog, TextDialog and its specialization InputFileDialog) are essentially little windows that pop up in the middle of the screen displaying a message and an editable subwindow. A NumberDialog works with an EditNum subwindow, while a TextDialog works with an EditText subwindow.

They have constructors (the constructor sets the message string and, in the case of the NumberDialog limits on the range permitted for input values) and a PoseModally() function. The PoseModally() function takes an input argument (a value to be displayed initially in the editable subwindow) and returns as a result (or as a result parameter) the input data received.

The PoseModally() function arranges for the editable subwindow to "get input" and performs other housekeeping, e.g. the InputFileDialog may try to open a file with the name entered by the user.

The class declarations are:

```
class NumberDialog : public EditWindow {
public:
    NumberDialog(const char* msg,
           long min = LONG_MIN, long max = LONG_MAX);
    long   PoseModally(long current);
protected:
    EditNum *fE;
};

class TextDialog : public EditWindow {
public:
    TextDialog(const char* msg);
    virtual void  PoseModally(char *current,
           char newdata[]);
protected:
    EditText *fE;
};

class InputFileDialog : public TextDialog {
public:
    InputFileDialog();
    void PoseModally(char *current, char newdata[],
           int checked = 1);
};
```

RecordWindow

As noted earlier, class RecordWindow is a slightly more sophisticated version of the same idea as embodied in class MenuWindow. Rather than use something specific like a set of "menu items", a RecordWindow utilizes its list of "subwindows". "Tabbing" in a MenuWindow moves a marker from one item to another; "tabbing" in a RecordWindow results in different editable subwindows being given the chance to "get input".

The class declaration is:

```
class RecordWindow : public EditWindow {
public:
    RecordWindow(Record *r);
    void   PoseModally();
protected:
    void           CountEditWindows();
    void           NextEditWindow();
    virtual void   InitEditWindows();

    Record         *fRecord;
    int            fNumEdits;
    int            fCurrent;
    EditWindow     *fEWin;
};
```

Auxiliary functions for PoseModally

Data members

The data members include the link to the associated Record, a count of the number of editable subwindows, an integer identifying the sequence number of the current editable subwindow and a pointer to that subwindow. (Pointers to subwindows are of course stored in the fSubWindows data member as declared in class Window.)

The constructor for class RecordWindow simply identifies it as something associated with a Record. The RecordWindow constructor makes it a full size (70x20) window.

The only additional public member function is PoseModally(). This is the function that arranges for each editable subwindow to have a turn at getting input. The function is defined as follows:

```
void RecordWindow::PoseModally()
{
    char ch;
    DisplayWindow();
    CountEditWindows();
    InitEditWindows();
    fCurrent = fNumEdits;

    do {

           NextEditWindow();
           fRecord->SetDisplayField(fEWin);


           ch = fEWin->GetInput();

           if(fEWin->ContentChanged()) {
                  fRecord->ReadDisplayField(fEWin);
                  fRecord->ConsistencyUpdate(fEWin);
                  }
    } while(ch != '\n');
}
```

Initialization

Loop until user ends input with "enter'

Activate next edit subwindow

Subwindow gets input

Update record

The initialization steps deal with things like getting the window displayed and determining the number of editable subwindows.

The main body of the `PoseModally()` function is its loop. This loop will continue execution until the user terminates all input with the "enter" key.

The code of the loop starts with calls to an auxiliary private member function that picks the "next" editable subwindow. The associated `Record` object is then told to (re)initialize the value in the subwindow. Once its contents have been reset to correspond to those in the appropriate member of the `Record`, the edit window is given its chance to "get input".

The editable item will return the character that stopped its "get input" loop. This might be the `'\n'` ("enter") character (in which case, the driver loop in `PoseModally()` can finish) or it might be a "tab" character (or any other character that the edit window can not handle, e.g. a `'*'` in an `EditNum`).

When an editable subwindow has finished processing input, it is asked whether its value has changed. If the value has been changed, the associated `Record` object is notified. A `Record` will have to read the new value and copy it into the corresponding data member. Sometimes, there is additional work to do (the "consistency update" call).

Example trace of interactions

Figure 30.9 illustrates a pattern of interactions between a `RecordWindow` and a `Record`. The example shown is for a `StudentRec` (as shown in Figure 30.1). It illustrates processes involved in changing the mark for assignment 1 from its default 0 to a user specified value.

When the loop in `RecordWindow::PoseModally()` starts, the `EditText` subwindow for the student's name will become the active subwindow. This results in the first interactions shown. The `RecordWindow` would ask the `Record` to set that display field; the `Record` would invoke `EditText::SetVal()` to set the current string. Then, the `EditText` object would be asked to `GetInput()`.

If the user immediately entered "tab", the `GetInput()` function would return leaving the `EditText` object unchanged.

After verifying that the subwindow's state was unchanged, the `RecordWindow` would arrange to move to the next subwindow. This would be the `EditNum` with the mark for the first assignment.

The `Record` would again be asked to set a subwindow's initial value. This would result in a call to `EditNum::SetVal()` to set the initial mark.

The `EditNum` subwindow would then have a chance to `GetInput()`. It would loop accepting digits (not shown in Figure 30.9) and would calculate the new value.

When this input step was finished, the `RecordWindow` could check whether the value had been changed and could get the `Record` to deal with the update.

30.6.2 Implementation Code

This section contains example code for the implementation of the window classes. Not all functions are given, but the code here should be sufficient to allow implementation. (The complete code can be obtained by ftp over the Internet as explained in the preface.)

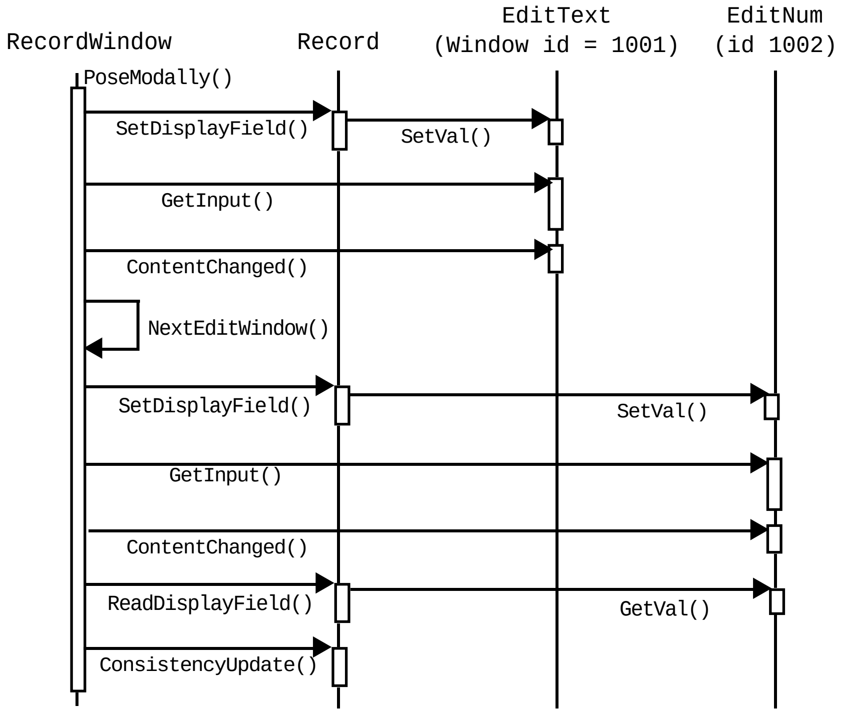


Figure 30.9 Example trace of specific object interactions while a RecordWindow is "posed modally".

Window

Most of the code is similar to that given in Chapter 29. Extra functions like `ShowText()` and `ShowNumber()` should be straightforward to code.

The `DisplayWindow()` function gets the contents of a window drawn, then arranges to get each subwindow displayed:

```
void Window::DisplayWindow()
{
    PrepareContent();
    PrepareToDisplay();
    ShowAll();
    int n = fSubWindows.Length();
    for(int i=1; i<=n; i++) {
           Window* sub = (Window*) fSubWindows.Nth(i);
           sub->DisplayWindow();
           }
}
```

The destructor will need to have a loop that removes subwindows from the `fSubWindows` collection and deletes them individually.

EditWindow

The constructor for class `EditWindow` passes most of the arguments on to the constructor of the `Window` base class. Its only other responsibility is to set the "validation" flag.

```
EditWindow::EditWindow(int id, int x, int y, int width,
                  int height, int mustValidate)
    : Window(id, x, y, width, height)
{
    fMustValidate = mustValidate;
}
```

The `GetInput()` function implements the algorithm given earlier. Characters are obtained by invoking the `GetChar()` function of the `WindowRep` object.

Normal "get input" behaviour

```
char EditWindow::GetInput()
{
    int v;
    char ch;
    InitializeInput();
    do {
           SetCursorPosition();
           fEntry = 0;
           ch = WindowRep::Instance()->GetChar();
           while(ch != '\n') {
                  if(!HandleCharacter(ch))
                         break;
                  ch = WindowRep::Instance()->GetChar();
                  }
           TerminateInput();
           v = Validate();
    } while (fMustValidate && !v);
    return ch;
}
```

EditText

The `EditText` constructor is mainly concerned with sorting out the width of any label and getting the label copied into the "background" image array (via the call to `ShowText()`). The arguments passed to the `EditWindow` base class constructor fix the height of an `EditText` to three rows (content plus frame).

```
EditText::EditText(int id, int x, int y, int width,
    char *label,  short size, int mustValidate)
           : EditWindow(id, x, y, width, 3, mustValidate)
{
    fSize = size;
    fLabelWidth = 0;
    int s = (label == NULL) ? 2 : strlen(label)+2;
    width -= 6;
    fLabelWidth = (s < width) ? s : width;
    ShowText(label, 2, 2, fLabelWidth);
    fBuf[0] = '\0';
    fChanged = 0;
}
```

The `SetVal()` member function copies the given string into the `fBuf` array (avoiding any overwriting that would occur if the given string was too long):

```
void EditText::SetVal(char* val, int redraw)
{
    int n = strlen(val);
    if(n>254) n = 254;
    strncpy(fBuf,val,n);
    fBuf[n] = '\0';
    ShowValue(redraw);
    fChanged = 0;
}
```

The `ShowValue()` member function outputs the string to the right of any label already displayed in the `EditText`. The output area is first cleared out by filling it with spaces and then characters are copied from the `fBuf` array:

EditText::ShowValue()

```
void EditText::ShowValue(int redraw)
{
    int left = fLabelWidth;
    int i,j;
    for(i=left; i<fWidth;i++)
                  fCurrentImg[1][i-1] = ' ';
    for(i=left,j=0; i<fWidth; i++, j++) {
           char ch = fBuf[j];
           if(ch == '\0') break;
           fCurrentImg[1][i-1] = ch;
           }
    if(redraw)
           ShowContent();
}
```

If the string is long, only the leading characters get shown.

Function `EditText::HandleCharacter()` accepts all characters apart from control characters (e.g. "enter") and the "tab" character:

Handling input characters

Return "fail" if unacceptable character

Initialization for first character

Store character

```
int EditText::HandleCharacter(char ch)
{
    if(iscntrl(ch) || (ch == kTABC))
           return 0;

    if(fEntry == 0) {
           ClearArea(fLabelWidth, 2, fWidth-1, 2,0);
           Set(fLabelWidth, 2, ch);
           SetPromptPos(fLabelWidth+1, 2);
           }
    fBuf[fEntry] = ch;
    fChanged = 1;
    fEntry++;
```

```
        if(fEntry == fSize) return 0;
        else return 1;
    }
```

Normally, an `EditText` will start by displaying the default text. This should be cleared when the first acceptable character is entered. Variable `fEntry` (set to zero in `InitializeInput()`) counts the number of characters entered. If its value is zero a `ClearArea()` operation is performed. Acceptable characters fill out the `fBuf` array (until the size limit is reached).

The remaining `EditText` member functions are all trivial:

```
int EditText::ContentChanged() { return fChanged; }

void EditText::InitializeInput()
{
    EditWindow::InitializeInput();
    fEntry = 0;
}

void EditText::SetCursorPosition()
{
    SetPromptPos(fLabelWidth, 2);
}

void EditText::TerminateInput()
{
    fBuf[fEntry] = '\0';
}
```

EditNum

Class `EditNum` is generally similar to class `EditText`. Again, its constructor is mainly concerned with sorting out the width of any label

```
EditNum::EditNum(int id, int x, int y, int width,
    char *label, long min, long max, int mustValidate)
            : EditWindow(id, x, y, width, 3, mustValidate)
{
    fMin = min;
    fMax = max;;
    fSetVal = fVal = 0;
    int s = (label == NULL) ? 2 : strlen(label)+2;
    width -= 6;
    fLabelWidth = (s < width) ? s : width;
    ShowText(label, 2, 2, fLabelWidth);
}
```

It also sets the data members that record the allowed range of valid inputs and sets the "value" data member to zero.

The `SetVal()` member function restricts values to the permitted range:

```
void EditNum::SetVal(long val, int redraw)
```

```
{
    if(val > fMax) val = fMax;
    if(val < fMin) val = fMin;
    fSetVal = fVal = val;
    ShowValue(redraw);
}
```

Member `ShowValue()` has to convert the number to a string of digits and get these output (if the number is too large to be displayed it is replaced by '#' marks). There is a slight inconsistency in the implementation of `EditNum`. The initial value is shown "right justified" in the available field. When a new value is entered, it appears "left justified" in the field. (Getting the input to appear right justified is not really hard, its just long winded. As successive digits are entered, the display field has to be cleared and previous digits redrawn one place further left.)

EditNum:: ShowValue()

```
void EditNum::ShowValue(int redraw)
{
    int left = fLabelWidth;
    int pos = fWidth - 1;
    long val = fVal;
    for(int i = left; i<= pos; i++)
           fCurrentImg[1][i-1] = ' ';
    if(val<0) val = -val;
    if(val == 0)
           fCurrentImg[1][pos-1] = '0';
    while(val > 0) {
           int d = val % 10;
           val = val / 10;
           char ch = d + '0';
           fCurrentImg[1][pos-1] = ch;
           pos--;
           if(pos <= left) break;
           }
    if(pos<=left)
           for(i=left; i<fWidth;i++)
                  fCurrentImg[1][i-1] = '#';
    else
    if(fVal<0)
           fCurrentImg[1][pos-1] = '-';
    if(redraw)
           ShowContent();
}
```

Clear field

Generate digit string starting at right

"Hash fill" when number too large

Add sign when needed

The `HandleCharacter()` member function has to deal with a couple of special cases. An input value may be preceded by a + or - sign character; details of the sign have to be remembered. After the optional sign character, only digits are acceptable. The first digit entered should cause the display field to be cleared and then the digit should be shown (preceded if necessary by a minus sign).

EditNum:: HandleCharacter()

```
int EditNum::HandleCharacter(char ch)
{
    // ignore leading plus sign(s)
    if((fEntry == 0) && (ch == '+'))
           return 1;
```

Deal with initial sign

```
    if((fEntry == 0) && (ch == '-')) {
           fsign = -fsign;
           return 1;
           }
```

Terminate input on non-digit

```
    if(!isdigit(ch))
           return 0;
    if(fEntry == 0) {
```

Clear entry field for first digit

```
           ClearArea(fLabelWidth, 2, fWidth-1, 2, 0);
           fVal = 0;
           if(fsign<0) {
                  Set(fLabelWidth, 2, '-');
                  Set(fLabelWidth+1, 2, ch);
                  }
           else Set(fLabelWidth, 2, ch);
           }
```

Consume ordinary digits

```
    fEntry++;
    fVal = fVal*10 + (ch - '0');
    return 1;
}
```

As the digits are entered, they are used to calculate the numeric value which gets stored in `fVal`.

The ± sign indication is held in `fsign`. This is initialized to "plus" in `InitializeInput()` and is used to set the sign of the final number in `TerminateInput()`:

```
void EditNum::InitializeInput()
{
    EditWindow::InitializeInput();
    fsign = 1;
}

void EditNum::TerminateInput()
{
    fVal = fsign*fVal;
}
```

If the number entered is out of range, the `Validate()` function fills the entry field with a message and then, using the `Beep()` and `Delay()` functions of the `WindowRep` object, brings error message to the attention of the user:

EditNum::Validate

```
int EditNum::Validate()
{
    if(!((fMin <= fVal) && (fVal <= fMax)) ){
```

Invalid entry

```
           ShowText("(out of range)", fLabelWidth , 2,
                  fWidth-1,0,0);
           WindowRep::Instance()->Beep();
           WindowRep::Instance()->Delay(1);
           fVal = fSetVal;
           ClearArea(fLabelWidth, 2, fWidth-1, 2, 0);
           ShowValue(1);
```

Failure return

```
           return 0;
           }
```

```
    else return 1;
}
```

The remaining member functions of class `EditNum` are simple:

```
int EditNum::ContentChanged() { return fVal != fSetVal; }

void EditNum::SetCursorPosition()
{
    SetPromptPos(fLabelWidth, 2);
}
```

MenuWindow

The constructor for class `MenuWindow` initializes the count of menu items to zero and adds a line of text at the bottom of the window:

```
MenuWindow::MenuWindow(int id, int x, int y, int width, int
height)
    : EditWindow(id, x, y, width, height)
{
    fChoices = 0;
    ShowText("(use 'option-space' to switch between
           "choices, 'enter' to select)",
           2, height-1, width-4);
}
```

Menu items are "added" by writing their text into the window background (the positions for successive menu items are predefined). The menu numbers get stored in the array `fCmds`.

MenuWindow:: AddMenuItem()

```
int MenuWindow::AddMenuItem(const char *txt, int num)
{
    if(fChoices == kMAXCHOICES)
           return 0;
    int len = strlen(txt);
    fCmds[fChoices] = num;
    int x = 10;
    int y = 4 + 3*fChoices;
    ShowText(txt, x, y, fWidth-14);
    fChoices++;
    return 1;
}
```

The `PoseModally()` member function gets the window displayed and then uses `GetInput()` (as inherited from class `EditWindow`) to allow the user to select a menu option:

MenuWindow:: PoseModally()

```
int MenuWindow::PoseModally()
{
    DisplayWindow();
    fChosen = 0;
    GetInput();
    return fCmds[fChosen];
}
```

The "do nothing" function `EditWindow::HandleCharacter()` is redefined. Function `MenuWindow::HandleCharacter()` interprets "tabs" as commands changing the chosen item.

***MenuWindow::
HandleCharacter()***

```
int MenuWindow::HandleCharacter(char ch)
{
```

Ignore anything except tabs

```
    if(ch != kTABC)
           return 1;
```

Clear cursor

```
    ClearCursorPosition();
```

Update chosen item

```
    fChosen++;
    if(fChosen==fChoices)
           fChosen = 0;
    SetCursorPosition();
    return 1;
}
```

(The "tab" character is defined by the character constant `kTABC`. On some systems, e.g. Symantec's environment, actual tab characters from the keyboard get filtered out by the run-time routines and are never passed through to a running program. A substitute tab character has to be used. One possible substitute is "option-space" (`const char kTABC = 0xCA;`).

The auxiliary member functions `SetCursorPosition()` and `ClearCusorPostion()` deal with the display of the ==> cursor indicator:

```
void MenuWindow::SetCursorPosition()
{
    int x = 5;
    int y = 4 + 3*fChosen;
    ShowText("==>", x, y, 4,0,0);
    SetPromptPos(x,y);
}

void MenuWindow::ClearCursorPosition()
{
    int x = 5;
    int y = 4 + 3*fChosen;
    ShowText("   ", x, y, 4, 0, 0);
}
```

Dialogs

The basic dialogs, `NumberDialog` and `TextDialog`, are very similar in structure and are actually quite simple. The constructors create an `EditWindow` containing a prompt string (e.g. "Enter record number"). This window also contains a subwindow, either an `EditNum` or an `EditText`.

Dialog constructors

```
NumberDialog::NumberDialog(const char *msg,
           long min, long max)
    : EditWindow(kNO_ID, 15, 5, 35,10)
{
    fE = new EditNum(kNO_ID, 5, 5, 20, NULL, min, max, 1);
    ShowText(msg, 2, 2, 30);
    AddSubWindow(fE);
}

TextDialog::TextDialog(const char *msg)
    : EditWindow(kNO_ID, 15, 5, 35,10)
{
    fE = new EditText(kNO_ID, 5, 5, 20, NULL, 63, 1);
    ShowText(msg, 2, 2, 30);
    AddSubWindow(fE);
}
```

The `PoseModally()` functions get the window displayed, initialize the editable subwindow, arrange for it to handle the input and finally, return the value that was input.

PoseModally() functions

```
long NumberDialog::PoseModally(long current)
{
    DisplayWindow();
    fE->SetVal(current, 1);
    fE->GetInput();
    return fE->GetVal();
}

void TextDialog::PoseModally(char *current, char newdata[])
{
    DisplayWindow();
    fE->SetVal(current, 1);
    fE->GetInput();
    strcpy(newdata,fE->GetVal());
}
```

An `InputFileDialog` is simply a `TextDialog` whose `PoseModally()` function has been redefined to include an optional check on the existence of the file:

InputFileDialog

```
InputFileDialog::InputFileDialog() :
    TextDialog("Name of input file")
{
}

void InputFileDialog::PoseModally(char *current,
    char newdata[], int checked)
{
    DisplayWindow();
    for(;;) {
```

Loop until valid file name given

```
                fE->SetVal(current, 1);
                fE->GetInput();
                strcpy(newdata,fE->GetVal());
                if(!checked)
                       return;
```

Try opening file

```
                ifstream in;
                in.open(newdata,ios::in | ios::nocreate);
                int state = in.good();
                in.close();
                if(state)
                       return;
```

Warn of invalid file name

```
                WindowRep::Instance()->Beep();
                fE->SetVal("File not found", 1);
                WindowRep::Instance()->Delay(1);
                }
}
```

The `Alert()` function belongs with the dialogs. It displays an `EditWindow` with an error message. This window remains on the screen until dismissed by the using hitting the enter key.

Global function Alert()

```
void Alert(const char *msg)
{
    EditWindow e(kNO_ID, 15, 5, 35, 10);
    e.DisplayWindow();
    e.ShowText(msg, 2, 2, 30, 0, 0);
    e.ShowText("OK", 18, 6, 3, 0, 0);
    e.GetInput();
}
```

RecordWindow

As far as the constructor is concerned, a `RecordWindow` is simply an `EditWindow` with an associated `Record`:

```
RecordWindow::RecordWindow(Record *r)
    : EditWindow(0, 1, 1, 70, 20)
{
    fRecord = r;
}
```

RecordWindow:: PoseModally()

The main role of a `RecordWindow` is to be "posed modally". While displayed arranges for its subwindows to "get input". (These subwindows get added using the inherited member function `Window::AddSubWindow()`). The code for `Record Window::PoseModally()` was given earlier.

The auxiliary functions `CountEditWindows()`, `InitEditWindows()`, and `NextEditWindow()` work with the subwindows list. The `CountEditWindows()` function runs through the list of subwindows asking each whether it "can handle input":

```
void RecordWindow::CountEditWindows()
{
    fNumEdits = 0;
    int nsub = fSubWindows.Length();
    for(int i = 1; i <= nsub; i++) {
           Window* w = (Window*) fSubWindows.Nth(i);
           if(w->CanHandleInput())
                  fNumEdits++;
           }
}
```

Function `InitEditWindows()`, called when the `RecordWindow` is getting displayed, arranges for the associated `Record` to load each window with the appropriate value (taken from some data member of the `Record`):

```
void RecordWindow::InitEditWindows()
{
    int nsub = fSubWindows.Length();
    for(int i = 1; i <= nsub; i++) {
           Window* w = (Window*) fSubWindows.Nth(i);
           if(w->CanHandleInput())
                  fRecord->SetDisplayField((EditWindow*)w);
           }
}
```

Function `NextEditWindow()` updates the value of `fCurrent` (which identifies which editable subwindow is "current"). The appropriate subwindow must then be found (by counting through the `fSubWindows` list) and made the currently active window that will be given the chance to "get input".

```
void RecordWindow::NextEditWindow()
{
    if(fCurrent == fNumEdits)
           fCurrent = 1;
    else
           fCurrent++;
    int nsub = fSubWindows.Length();
    for(int i = 1, j= 0; i <= nsub; i++) {
           Window* w = (Window*) fSubWindows.Nth(i);
           if(w->CanHandleInput()) {
                  j++;
                  if(j == fCurrent) {
                         fEWin = (EditWindow*) w;
                         return;
                         }
                  }
           }
}
```

30.6.3 Using the concrete classes from a framework

The `Window` class hierarchy has been presented in considerable detail.

Here, the detail was necessary. After all, you are going to have to get the framework code to work for the exercises. In addition, the structure and implementation code illustrate many of the concepts underlying elaborate class hierarchies.

Apparent complexity

Class like `InputFileDialog` or `EditNum` are actually quite complex entities. They have more than thirty member functions and between ten and twenty data members. (There are about 25 member functions defined for class `Window`; other functions get added, and existing functions get redefined at the various levels in the hierarchy like `EditWindow`, `TextDialog` etc).

Simplicity of use

This apparent complexity is not reflected in their use. As far as usage is concerned, an `InputFileDialog` is something that can be asked to get the name of an input file. The code using such an object is straightforward, e.g.

```
void Document::OpenOld()
{
    InputFileDialog oold;
    oold.PoseModally("example",fFileName, fVerifyInput);
    OpenOldFile();
}
```

(*"Give me an InputFileDialog". "Hey, file dialog, do your stuff".*)

This is typical. Concrete classes in the frameworks may have complex structures resulting from inheritance, but their use is simple.

In real frameworks, the class hierarchies are much more complex. For example, one of the frameworks has a class `TScrollBar` that handles scrolling of views displayed in windows. It is basically something that responds to mouse actions in the "thumb" or the "up/down" arrows and it changes a value that is meant to represent the origin that is to be used when drawing pictures. Now, a `TScrollBar` is a kind of `TCtlMgr` which is a specialization of `TControl`. Class `TControl` is a subclass of `TView`, a `TView` is derived from `TCommandHandler`. A `TCommandHandler` is actually a specialization of class `TEventHandler`, and, naturally, class `TEventHandler` is derived from `TObject`.

By the time you are six or seven levels deep in a class hierarchy, you have something fairly complex. A `TScrollBar` will have a fair number of member functions defined (about 300 actually) because it can do anything a `TObject` can do, and all the extra things that a `TEventHandler` added, plus the behaviours of a `TCommandHandler`, while `TView` and others added several more abilities.

Usually, you aren't interested. When using a `TScrollBar`, all you care is that you can create one, tell it to do its thing, and that you can use a member function `GetVal()` to get the current origin value when you need it.

The reference manuals for the class libraries will generally document just the unique capabilities of the different concrete classes. So something like an `InputFileDialog` will be described simply in terms of its constructor and `PoseModally()` function.

Of course sometimes you do make some use of functions that are inherited from base classes. The `RecordWindow` class is an example. This could be described primarily in terms of its constructor, and `PoseModally()` functions. However, code using a `RecordWindow` will also use the `AddSubWindows()` member function.

30.7 ORGANIZATIONAL DETAILS

When working with a framework, your primary concerns are getting to understand the conceptual application structure that is modelled in the framework, and the role of the different classes.

However, you must also learn how to build a program using the framework. This is a fairly complex process, though the integrated development environments may succeed in hiding most of the complexities.

Problems with "header" files and linking of compiled modules

There are a couple of sources of difficulties. Firstly, you have all the "header" files with the class declarations. When writing code that uses framework classes, you have to #include the appropriate headers. Secondly, there is the problem of linking the code. If each of the classes is in a separate implementation file, you will have to link your compiled code with a compiled version of each of the files that contains framework code that you rely on.

Headers

"The enormous header file"

One way of dealing with the problem of header files is in effect to #include them all. This is normally done by having a header file, e.g. "ClassLib.h", whose contents consist of a long sequence of #includes:

```
#include "Cmdhdl.h"
#include "Application.h"
#include "Document.h"
#include "WindowRep.h"
…
#include "Dialog.h"
```

This has the advantage that you don't have to bother to work out which header files need to be included when compiling any specific implementation (.cp) file.

Slow compilations, large memory usage

The disadvantage is that the compiler has to read all those headers whenever a piece of code is compiled. Firstly, this makes the compilation process slow. With a full size framework, there might be fifty or more header files; opening and reading the contents of fifty files takes time. Secondly, the compiler has to record the information from those files in its "symbol tables". A framework library may have one hundred or more classes; these classes can have anything from ten to three hundred member functions. Recording this information takes a lot of space. The compiler will need many megabytes of storage for its symbol tables (and this had better be real memory, not virtual memory on disk, because otherwise the process becomes too slow).

Interdependencies among class declarations

There are other problems related to the headers, problems that have to be sorted out by the authors of the framework. For example, there are dependencies between different classes and these have to be taken into account when arranging declarations in a header file. In a framework, these dependencies generally appear as the occurrence of data members (or function arguments) that are pointers to instances of other classes, e.g:

```
class Record {
    …
    RecordWindow  *fWin;
    …
};

class RecordWindow {
…
    Record *fRec;
…
};
```

If the compiler encounters class `Record` before it learns about class `RecordWindow` it will complain that it does not understand the declaration `RecordWindow *fWin`. If the compiler encounters class `RecordWindow` before it learns about class `Record` it will complain that it doesn't understand the declaration `Record *fRec`.

One solution to this problem is to have dummy class declarations naming the classes before any instances get mentioned:

```
class CommandHandler;
class Application;
class Document;
class Window;
class Record;
class RecordWindow;
…
// Now get first real class declaration

class CommandHandler {
…
};
```

Such a list might appear at the start of the composite "ClassLib.h" file.

An alternative mechanism is to include the keyword `class` in all the member and argument declarations:

```
class RecordWindow {
public:
    RecordWindow(class Record *r);
    …
protected:
    …
    class Record  *fRec;
    …
};
```

Linking

The linking problem is that a particular program that uses the framework must have the code for all necessary framework classes linked to its own code.

One way of dealing with this is, once again, to use a single giant file. This file contains the compiled code for all framework classes. The "linker" has to scan this file to find the code that is needed.

Such a file does tend to be very large (several megabytes) and the linkage process may be quite slow. The linker may have to make multiple passes through the file. For example, the linker might see that a program needs an `InputFileDialog`; so it goes to the library file and finds the code for this class. Then it finds that an `InputFileDialog` needs the code for `TextDialog`; once again, the linker has to go and read through the file to find this class. In unfortunate cases, it might be necessary for a linker to read the file three or four times (though there are alternative solutions).

If the compiled code for the various library classes is held in separate files, the linker will have to be given a list with the names of all these files.

Currently the Symantec system has a particularly clumsy approach to dealing with the framework library. In effect, it copies all the source files of the framework into each "project" that is being built using that framework. Consequently, when you start, you discover that you already have 150 files in your program. This leads to lengthy compilation steps (at least for the first compilation) as well as wastage of disk space. (It also makes it easier for programmers to change the code of the framework; changing the framework is not a wise idea.)

RecordFile Framework

Figure 30.10 illustrates the module structure for a program built using the RecordFile framework. Although this example is more complex than most programs that you would currently work with, it is typical of real programs. Irrespective of whether you are using OO design, object based design, or functional decomposition, you will end up with a program composed from code in many implementation files. You will have header files declaring classes, structures and functions. There will be interdependencies. One of your tasks as a developer is sorting out these dependencies.

The figure shows the various files and the #include relationships between files. Sometimes a header file is #included within another header; sometimes a header file is #included by an implementation (.cp) file.

When you have relationships like class derivation, e.g. `class MyApp : public Application`, the header file declaring a derived class like `MyApp` (My.h) has to #include the header defining the base class `Application` (Application.h).

If the relationship is simply a "uses" one, e.g. an instance of class `RecordWin` uses an instance of class `Record`, then the #include of class `Record` can be placed in the RecordWin.cp file (though you will need a declaration like `class Record` within the RecordWin.h file in order to declare `Record*` data members).

The files used are as follows:

- commands.h
  Declares constants for "command numbers" (e.g. `cNEW` etc). Used by all `CommandHandler` classes so #included in CmdHdl.h

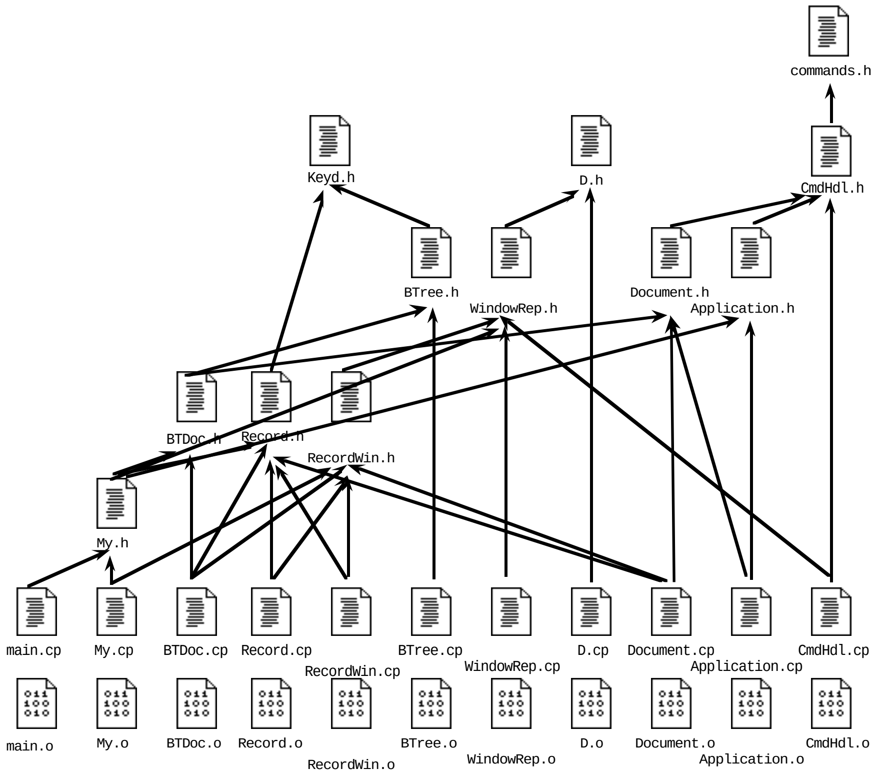


Figure 30.10 Module structure for a program using the RecordFile framework.

- Keyd.h
  Declares pure abstract class `KeyedStorableItem`. This has to be #included by BTree.h and Record.h. (The class declaration in Record.h tells the compiler that a `Record` is a `KeyedStorableItem` so we had better let the compiler know what a `KeyedStorableItem` is.)

- D.h
  Declares the dynamic array. A `Window` contains an instance of class `DynamicArray` so the header D.h will have to be #included in WindowRep.h.

- CmdHdl.h
  Declares the abstract base class `CommandHandler`.

- BTree.h
  Declares class BTree.

- WindowRep.h
  Declares classes: `WindowRep`, `Window`, `NumberItem`, `EditWindow`, `EditText`, `EditNum`, and the "dialog" classes. A program that uses the window code typically uses all different kinds of windows so they might as well be declared and define as a group.

- Document.h and Application.h
  Declare the corresponding "partially implemented abstract classes".

- BTDoc.h
  Declares class BTDoc, #including the Document.h class (to get a definition of the base class Document), and BTree.h to get details of the storage structure used.

- Record.h and RecordWin.h
  Declare the `Record` and `RecordWin` classes. Class `RecordWin` is separate from the other Window classes. The main group of Window classes could be used in other programs; `RecordWin` is special for the type of programs considered in this chapter.

- My.h
  Declares the program specific "MyApp", "MyDocument", and "MyRec" classes.

- The ".cp" implementation files
  The code with definitions for the classes, additional #includes as needed.

The main program, in file main.cp, will simply declare an instance of the "MyApp" class and tell it to "run".

30.8 THE "STUDENTMARKS" EXAMPLE PROGRAM

We might as well start with the main program. It has the standard form:

```
int main()
{
    StudentMarkApp a;
    a.Run();
    return 0;
}
```

The three classes that have to be defined are `StudentMarkApp`, `StudentMarkDoc`, and `StudentRec`. The "application" and "document" classes are both simple:

Application and Document classes

```
class StudentMarkApp : public Application {
protected:
    virtual       Document* DoMakeDocument();
};

class StudentMarkDoc : public BTDoc {
protected:
    virtual Record       *DoMakeRecord(long recnum);
    virtual Record       *MakeEmptyRecord();
};
```

The implementation for their member functions is as expected:

Implementation of Application and Document classes

```
Document *StudentMarkApp ::DoMakeDocument()
{
    return new StudentMarkDoc;
}

Record *StudentMarkDoc::DoMakeRecord(long recnum)
{
    return new StudentRec (recnum);
}

Record *StudentMarkDoc::MakeEmptyRecord()
{
    return new StudentRec (0);
}
```

Record class

The specialized subclass of class `Record` does have some substance, but it is all quite simple:

```
class StudentRec : public Record {
public:
    StudentRec (long recnum);

    virtual void  WriteTo(fstream& out) const;
    virtual void  ReadFrom(fstream& in);
    virtual long  DiskSize(void) const ;

    virtual void  SetDisplayField(EditWindow *e);
    virtual void  ReadDisplayField(EditWindow *e);
protected:
    virtual void  ConsistencyUpdate(EditWindow *e);
    virtual void  AddFieldsToWindow();
    char   fStudentName[64];
    long   fMark1;
    long   fMark2;
    long   fMark3;
    long   fMark4;
    long   fMidSession;
    long   fFinalExam;

    NumberItem    *fN;
    long   fTotal;
};
```

Redefine KeyedStorableItem functions

Redefine Record functions

Declare unique data members

The data required include a student name and marks for various assignments and examinations. There is also a link to a `NumberItem` that will be used to display the total mark for the student.

The constructor does whatever an ordinary `Record` does to initialize itself, then sets all its own data members:

```
StudentRec::StudentRec(long recnum) : Record(recnum)
{
    fMark1 = fMark2 = fMark3 = fMark4 =
           fMidSession = fFinalExam = 0;
```

```
        strcpy(fStudentName,"Nameless");
    }
```

The function `AddFieldsToWindow()` populates the `RecordWindow` with the necessary `EditText` and `EditNum` editable subwindows:

Building the display structure

Adding an EditText

Adding some EditNum subwindows

Adding a NumberItem

```
    void StudentRec::AddFieldsToWindow()
    {
        Record::AddFieldsToWindow();

        EditText *et = new EditText(1001, 5, 4, 60,
                "Student Name ");
        fRW->AddSubWindow(et);

        EditNum *en = new EditNum(1002, 5, 8, 30,
                "Assignment 1 (5) ", 0, 5,1);
        fRW->AddSubWindow(en);
        en = new EditNum(1003, 5, 10, 30,
                "Assignment 2 (10) ", 0, 10,1);
        fRW->AddSubWindow(en);
        …
        en = new EditNum(1007, 40, 10, 30, "Examination (50) ",
                0, 60,1);
        fRW->AddSubWindow(en);

        fTotal = fMark1 + fMark2 + fMark3 + fMark4
                     + fMidSession + fFinalExam;
        fN = new NumberItem(2000, 40, 14,30, "Total ", fTotal);
        fRW->AddSubWindow(fN);
    }
```

The call to the base class function, `Record::AddFieldsToWindow()`, gets the `NumberItem` for the record number added to the `RecordWindow`. An extra `NumberItem` subwindow is added to hold the total.

Functions `SetDisplayField()` and `ReadDisplayField()` transfer data to/from the `EditText` (for the name) and `EditNum` windows:

Data exchange with editable windows

Setting an EditText

Setting EditNum

```
    void StudentRec ::SetDisplayField(EditWindow *e)
    {
        long id = e->Id();
        switch(id) {
    case 1001:
                ((EditText*)e)->SetVal(fStudentName, 1);
                break;
    case 1002:
                ((EditNum*)e)->SetVal(fMark1,1);
                break;
    case 1003:
                ((EditNum*)e)->SetVal(fMark2,1);
                break;
    case 1004:
                ((EditNum*)e)->SetVal(fMark3,1);
                break;
    case 1005:
                ((EditNum*)e)->SetVal(fMark4,1);
```

```
           break;
case 1006:
           ((EditNum*)e)->SetVal(fMidSession,1);
           break;
case 1007:
           ((EditNum*)e)->SetVal(fFinalExam,1);
           break;
    }
}

void StudentRec::ReadDisplayField(EditWindow *e)
{
    long id = e->Id();
    switch(id) {
```

Reading an EditText

```
case 1001:
    char* ptr = ((EditText*)e)->GetVal();
    strcpy(fStudentName, ptr);
    break;
```

Reading an EditNum

```
case 1002:
    fMark1 = ((EditNum*)e)->GetVal();
    break;
…
…
case 1007:
    fFinalExam = ((EditNum*)e)->GetVal();
    break;
    }
}
```

The `ReadFrom()` and `WriteTo()` functions involve a series of low level read (write) operations that transfer the data for the individual data members of the record:

Reading from file

```
void StudentRec::ReadFrom(fstream& in)
{
    in.read((char*)&fRecNum, sizeof(fRecNum));
    in.read((char*)&fStudentName, sizeof(fStudentName));
    in.read((char*)&fMark1, sizeof(fMark1));
    in.read((char*)&fMark2, sizeof(fMark2));
    in.read((char*)&fMark3, sizeof(fMark3));
    in.read((char*)&fMark4, sizeof(fMark4));
    in.read((char*)&fMidSession, sizeof(fMidSession));
    in.read((char*)&fFinalExam, sizeof(fMidSession));
}
```

Writing to file

```
void StudentRec::WriteTo(fstream& out) const
{
    out.write((char*)&fRecNum, sizeof(fRecNum));
    out.write((char*)&fStudentName, sizeof(fStudentName));
    out.write((char*)&fMark1, sizeof(fMark1));
    …
    out.write((char*)&fFinalExam, sizeof(fMidSession));
}
```

(The total does not need to be saved, it can be recalculated.)

The DiskSize() function computes the size of a record as held on disk:

```
long StudentRec::DiskSize(void) const
{
    return sizeof(fRecNum) + sizeof(fStudentName)
    + 6 * sizeof(long);
}
```

Function `ConsistencyUpdate()` is called after an editable subwindow gets changed. In this example, changes will usually necessitate the recalculation and redisplay of the total mark:

```
void StudentRec::ConsistencyUpdate(EditWindow *e)
{
    fTotal = fMark1 + fMark2 + fMark3 + fMark4
            + fMidSession + fFinalExam;
    fN->SetVal(fTotal, 1);
}
```

EXERCISES

1. Implement all the code of the framework and the StudentMark application.

2. Create new subclasses of class Collection and class Document so that an AVL tree can be used for storage of data records.

3. Implement the Loans program using your AVLDoc and AVLCollection classes.

   The records in the files used by the Loans program are to have the following data fields:

   customer name (64 characters)
   phone number (long integer)
   total payments this year (long integer)
   an array with five entries giving film names (64 characters) and weekly charge
   for movies currently on loan.

   The program is to use a record display similar to that illustrated in Figure 30.3.

4. There is a memory leak. Find it. Fix it correctly.
   (Hint, the leak will occur with a disk based collection. It is in one of the Document functions. The "obvious" fix is wrong because it would disrupt memory based collection structures.)