The Graphic User Interface

of The AudioGraph Recorder

(PC version)

A thesis presented in partial fulfillment of the requirements

for the degree of

Master of Science in Computer Science at Massey University

Shao H. Nie

2001

Acknowledgements

I am indebted to Professor Chris Jesshope, my project supervisor, for his technical support throughout the project. Without the advice, comments and great patience from him, this thesis would not have been possible. I thank to him especially for his in depth analysis of some specific areas of the thesis, and for teaching me so much.

Honourable thanks to Mr. Chris Philips, for his advice on the history and design of GUI. And thanks go to the teachers, Mr. Nigel Perry, Mr. Paul Lyon, and Mr. Giovanni, for their opinions regarding programming languages. I would also like to thank Ms. Regina and Mr. Horia Slusanschi for providing some icon files for the project as well as advices on the Macintosh AudioGraph system.

I wish to also convey my gratitude to all the staff for lab computer support, and the secretaries of the computer science department for their help in time.

Finally, I thank my family for their support.

Abstract

With the popularity of the use of computers and the development of the Internet, many multimedia-authoring systems have been developed for computer-based teaching and learning. This is playing an increasingly important role in education. One authoring system is the *AudioGraph* project developed at Massey university of N.Z., which have been developed for recording audio-graphic presentation material for publication in an *html* reference environment, i.e. "on the web".

One of the tools in the *AudioGraph* project is the *AudioGraph Recorder*, which is a Macintosh application for recording or authoring web-based multimedia presentations. Due to the success of the publication of the *AudioGraph* application and the need of PC users, an *AudioGraph Recorder* for the PCs is required. This project is about the porting of the *AudioGraph Recorder* from the Macintosh platform to the PC platform.

First this project report explains the functionality of the *AudioGraph Recorder* (the Macintosh version), especially how the end users interact with the interface of the *AudioGraph Recorder*, and the corresponding state changes of the controls in the interface.

Then the report compares the development tools used in both platforms. The Macintosh version of the *AudioGraph Recorder* has been developed with the *PowerPlant* framework in *CodeWarrior* environment, but the PC version uses MFC framework in *Visual C++ 6.0*.

This report also describes in detail how the interface of the *AudioGraph Recorder* application was constructed with the MFC, and implementation of some functionality of the application. At the same time some internals of the MFC framework are discussed.

Table Contents

Chapter 1 introduction

Chapter 2 The analysis of the Macintosh AudioGraph Recorder interface

List of Acronyms

AWT	Abstract Windows Toolkit
API	Application Programming Interface
GUI	Graphics User Interface
JFC	Java foundation class
MDI	Multiple Document Interface
MFC	Microsoft foundation class
OO	Object-Oriented
OOUI	Object-Oriented User Interface
OS	Operation System
RTCI	Run-Time Class Information
RTTI	Run-Time Type Information
SA/SD	structured analysis and structured design
SDI	Single Document Interface
SDK	Software development Kit
UML	Unified Modelling Language

Chapter 1 Introduction

Computers as a powerful tool have been used for educational applications since the early 1960's. However, for the three decades the educational computing primarily focus on programming and producing computer–assisted instruction–the use of computer to provide teaching-related materials in the form of drills, tutorials and simulation etc.

In the 1980's, the primary use of the computer in the classroom shifted from a content delivery device to a learning tool due to the emergence of the personal computer and various types of application software. Educators and students began using the available software for word processing, desktop publishing, database, and telecommunication and graphing. They also began experimenting with a new generation of educational software with "intelligence", which encouraged critical thinking, problem solving decision making, and exploration.

The globalisation of information and communication is changing societal assumptions, especially those in the traditional education system since the 1990's. As the advent of the Internet and personal computers became more and more popular, the traditional teaching and learning in the classroom is encountering a big challenge. The challenge forces educators to assess the value of educational technologies, update their technological skills and blend new technologies into their instructional setting [B1].

Ever-increasing numbers of educational institutions are promoting and exploring instructional technologies, and developing or using technology-based educational products to enhance the teaching-learning process[B1]. Due to the World Wide Web, there is growing number of online learning courses available on the Web. This is becoming a very popular trend around the world. Using a search engine on the Web, thousands of online courses can be found, such as the course listings at the Web addresses:

- http://google.yahoo.com/bin/query?p=online+learing&hc=0&hs=0

- http://www.altavista.com/cgibin/query?pg=q&what=web&stype=wms&q=online+learing&poa=yhoo

1.1 Online learning

"Online learning" is based on the use of the World Wide Web to deliver course content and enable asynchronous communication between instructor and students [A1]. This also is called Web-based learning. As courseware is available on the Web, it can be accessed wherever a student is as long as he has an internet-ready computer.

Online learning is a flexible, convenient way to improve skills and productivity. It is immediate, as the course content required is available on the Web at anytime. There is no face-to-face communication with an instructor or other students in an online environment. The asynchronous communication between instructors and students is processed by e.g. using e-mails. Learners are able to study at their own pace and they are not required to attend scheduled classes. For students with different cultural backgrounds, online learning provides a convenient way for them to obtain a better and more thorough understanding of the course materials. Another advantage of online learning is that educators are able to combine the latest technologies into their course content, e.g. using animated diagram, multimedia etc.

The form of the course for the online learning is mostly delivered using hypertext in HTML in the browser of Netscape or Microsoft Internet Explorer. As multimedia is more vivid, sound and video are sometimes combined into the course, although it is limited by the low bandwidth of the Internet. To overcome the storage limit in Internet servers and the relatively low modem speed, some educator use "CD/Web hybrids"[C1], which is storing course contents onto CDs.

The hardware and software requirements of the learners are a personal computer with Internet connection and browser. Sometimes a browser plug-in software is also needed. A browser plug-in is an application software that is embedded in the browser of Netscape or Internet Explorer by using the standard interface functions of Netscape or Internet Explorer. When a student tries to open a corresponding document,

Netscape or Internet Explorer will check and find the plug-in software in its plug-in folder. The document's data will then appear in the browser, although it is actually controlled by the plug-in software.

Web-based learning is apparently attracting more and more different kinds of potential learners. As technology changes so fast, working adults are under pressure to gain the latest skills. This leads them to online learning because it suits their time needs, i.e. to be flexible. People with disabilities find sitting in front of computers in Web-based learning bring so much convenience and confidence. Even on campus, students have tended to study by themselves, and gradually teaching methods are changing to pay more attention to the improvement of the analysis ability of students. This factors all lead to the conclusion that the future of the Web-based learning is very prosperous. Up to now, however, the current use of Web-based learning has been very limited due to the lack of standards. Some institutes are exploring, experimenting and implementing the "Virtual Classroom" [C2].

1.2 Virtual classroom

A classroom can be defined as a communication system in which a group of people/users get together to discuss something they want to learn, with visuals (pictures, diagrams) and text provided to them to help in understanding. Walls surrounding the conventional classroom provide protection from outside noise and interference, which leads to a more effective learning process.

"A virtual classroom, on the other hand, is a system that provides the same opportunities for the teaching and learning process, beyond the physical limits of the traditional classroom's walls, thanks to the use of computer communication networks" [C3]. Due to the ubiquity and popularity of the Internet—particularly the World Wide Web—most virtual classroom implementations are Web-based.

To set up a system for a virtual classroom requires different kinds of technical skills and some policy schemes, such as graphical creation and manipulation, CGI-based programming, software assessment and evaluation, Java programming, and network

set-up and maintenance. Current experiments in virtual classroom systems mostly adopt the Sever-Client model. By the connection to the Sever through the Internet, a client can ask for services, such as registration, course contents, and feedback of questions and examinations. The course content delivered usually uses the HTML-based form.

Developing a virtual classroom will add a significant amount of work to the educator's daily routine, so the use of the proper authoring tools is significant to the educators. As most educators are not computer specialists, the authoring tools must be considered to be easy to use, has short learning curve and minimum cost. In particular, the following aspects are a measure of a good authoring tool:

- Easy to convert presentation to HTML;
- Easy to import, create and edit images;
- Easy to import, create and edit sound;
- Easy to import, create and edit graphics and animations;
- It's better to be able to import, create and edit videos. As video creation, processing and distribution normally require extensive and expensive hardware and software support, at present time it is still an option.

1.3 Reviews of some current authoring tools

A lot of authoring tools are available to provide rich choices for people now. Features and price of an authoring tool are always the concerns of users. Below some typical kinds of popular authoring tools are introduced without discussing the price issue.

First is a kind of presentation authoring tools, such as Microsoft PowerPoint, which provide a useful way to insert pictures and charts, and to organise text for the product of presentations. Due to the Microsoft office package being ubiquitous on personal computers, PowerPoint is still popular among teachers in schools delivering course materials to students. However it fails in multimedia.

Another type of authoring tool that is commonly used is web editor. FrontPage Edit 97[B2] is a successful software, which provides an integrated environment for creating a web site with little effort. It has some features, such as comment forms and searching. Its simple graphical interface and comprehensive wizards greatly simplify the process of setting up web sites. FrontPage is a very impressive graphical editor.

This kind of tools obviously reduces the time for creating web sits, but knowledge of HTML is still required for users. Sometime users may need to know programming languages such as CGI, Java script etc.

With hardware performance greatly improved, especially CPU speed, multimedia techniques are combined in authoring tools, such as Adobe LiveMotion[B3]. LiveMotion is for producing individual Web graphics or entire pages with interactivity, animation, and sound features. As it is actually a process of a kind of cartoon movie, which is not just aimed at educators, it requires time for user to get familiar with it, in particular in creating animation and sound. The learning and development process usually has long time curve.

Some universities and institutes are exploring a kind of education authoring tools by modelling how a lecture gives a lecture to students in classroom. This kind of authoring tools is still under development and improvement. One of the successful authoring tools is the Macintosh version of the AudioGraph application, which has been developed by Professor Chris Jesshope in Massey University of New Zealand[B4]. It provides a simple tool for teachers and student to use in virtual classroom environment. Its successful publication has drawn the attention of PC users, which leads to the implementation of the PC version AudioGraph system in this project.

1.4 The introduction of the AudioGraph system

The concepts and definitions for the AudioGraph system used in this report all are directly or indirectly taken from the articles written by Mr. Chris Jesshope, which are available from [C4][B5][B6].

The Audiograph project[B4] was developed by Professor Mr. Chris Jesshope, the head of the Computer science department of Massey University, Palmerston North of New Zealand.

It uses the model of "lecture theatre", which simulates the practical classroom education system, "in which a presenter arrives with a set of prepared slides and has a device for displaying those slides and a means for annotating them with hand-drawn text and diagrams. In addition voice is used to reinforce the slides and animations presented"[C4].

The goals for the AudioGraph project were to provide educators an easy-to-use and low-cost multimedia authoring tool to prepare multimedia course contents in a "virtual classroom" environment, and to provide learners a tool to "watch" the multimedia course contents in a browser of internet.

The tools of the AudioGraph system are used to create and playback multi-media presentations for web publishing. Web sites can be automatically generated from the material recorded. Components of AudioGraph presentations are embedded within web pages and play automatically on being activated. On the learners' side, the presentation can be controlled just like playing tape recorders [C4].

The AudioGraph tools consist of three parts:

- The AudioGraph recorder

  The recorder can import existing presentations (such as from PowerPoint), has hand drawing, highlight and sound recording functionality. It can also annotate a series actions made by the operator. The annotations can be edited later. It can generate a HTML file directly for publishing in the Web.

- The AudioGraph plug_in / The AudioGraph Player

  The AudioGraph plug_in is a browser plug-in for playback of AudioGraph web-based multimedia presentations. It uses the standard functions of the

Netscape plug-in interface. The AudioGraph Player is a standalone application for playback of web-base multimedia presentations. Both of them allow the user to start and stop the presentation, and to position playback from anywhere within the presentation. Progress is indicated using a progress bar [C4].

The tools currently available are a Macintosh version Recorder, a Macintosh version Player, and two Plug-ins (for Mac and PC platform). Because of the needs of PC users, a PC version Recorder is developed in this project.

The AudioGraph system is an authoring tool that brings together two typical kinds of users (educator and learner) in education systems. Educators use one of the tools (AudioGraph Recorder) to prepare lecture-style presentation. AudioGraph plug in or the AudioGraph Player is installed on the learner's computer, which is used for the learners to "watch" the presentation.

The application has been used in some universities and companies, and has proved to be a successful and mature application [C4][C5]. It has rich functionality and provides low-bandwidth streaming. It uses the Macintosh conventional user interface (GUI), which presents users with a consistent "feel and look".

1.5 Graphics User Interface (GUI)

User interface is not just a screen view but the whole environment in which a user communicates with a computer. According to Bennett's model of the user interface[C9] it shows:

- A conceptual model, representing the user's understanding of the user's task.
- A presentation language, the form in which the system presents information to the user.
- An action language, the mechanisms the designer provides to users to interact with the system
- An implementation model, the design of the software and hardware that provide the system's functional capabilities.

The user begins with a task or goal, based on the conceptual model of the task, and then plans and uses the action language to interact with a computer system. The system will response and process the user input. The feedback through the presentation language shows the result of the previous actions and provides guidance on what to do next.

1.5.1 The introduction of graphics user interface

During the evolution of the user interface, the traditional interface has been the command line interface, in which a user interacts with a system by inputting in a command line. The command line works in a straightforward way, but it has some problems such as:

- Users need to learn the interface, as the overall structure is not visible to them.
- Not easy to correct the typing error, sometime need to re-type the whole command line.
- If the screen is clear, user may not know how to do.
- In command entry, additional information may be desired especially when an error occurs.
- It is easy for users to feel bored with the dull environment.

In 1970 the Xerox Star workstation brought out the model of graphics user interface(GUI). At this stage the interface is involved with graphics presentation to end-users. The interface uses "icons" as a representation of objects being manipulated such as documents and programs. Users get better visual feels and it is easy to get into an application. It is "graphical" but not "object-based" (note: some people argue that this kind of interface is at least partially object-based)[A2]. The Microsoft Window Program Manager is a typical example. The icons represent files, applications and folders, and are organised into windows with a menu and toolbar. The interface can be directly manipulated by mouse clicking and pointing. Also a command can be involved for opening an application in a command line mode.

Modern interface makes icon-representing objects fully functional, which is called "object-based" interface, e.g. Windows 95. In spite of being used in a different platform, the object-based interface exhibits the common characteristics[A2]:

- The environment is represented graphically, using images of familiar objects.
- Icons represented objects can be dragged and dropped into other icons.
- Window can be scrolled and resized, and have menus and toolbars.
- Icon can be directly manipulated to show its properties and status.

The current technology used in interface design makes icon representation more "object-oriented". The term object-oriented user interface (OOUI) is used defined a kind of GUIs that use icons and other graphical techniques to represent "real object". Dave Collinns provided the definition [A2] for OOUI in conceptual level:

- The GUI is generally considered object-oriented, which brings the characteristics to represent what an end user needs to know[A2]:
- Users perceive and act on objects.
- Users can classify objects based on how they behave.
- In the context of what users are trying to do, all the interface objects fit together into a coherent overall representation.

OOUI causes tense debate about the current popular platforms, e.g. Windows 98, NT, and Mac OS 8.5 or later. Are their interfaces object-oriented? In the article by Diane Gartner[B7], it indicated that the current Windows and Macintosh OS interface are not "100%" object-oriented but at least object-based by an example in Window NT: if a user changes a folder name, Windows fails to keep track of a shortcut that points to a real object contained in the folder. I personally think it is a problem with the degrees of the "intelligence" of icon-representing objects. In other words, icon objects need to improve their functionality, but they can never act as a real object. Some people may confuse the external part (interface) of a system with the internal implementation part by using OO language.

1.5.2 GUI implementation Model

As the current programming for Windows (e.g. Winows 95, NT or later) and Macintosh (e.g. Mac OS) both use event-driven GUIs[B8], this section concentrates on the event model.

In the event model of a GUI, the input caused by the user's interaction, such as mouse movements, button presses and keystrokes that may be combined together in sequences, are treated as a stream of primitive events. The output is also a stream of events, which may directly respond to the input, such as displaying text and graphics, creating sound etc.

The event model by Green's formulation[C6] includes:

- *Events* are generated by devices or event handlers. In OO terms, these correspond to the message sent to objects.
- *Events* are put into the *event queues* that distribute events to event handlers. The destination handler for an event may be determined by a state variable such as the mouse pointer location, or by the registered recipient of certain events.
- *Event handlers* are responsible for receiving and processing events. When an event is received, corresponding routings for the recipient object are executed. During the process, new events may be generated.
- Corresponding to different classes, *templates* are used to describe the behaviour of event handlers.

Event-driven programming, such as Windows programming, has a fundamental difference with the traditional programming (e.g. DOS programs). In an event-driven model, the program is subservient and the OS is in control[A3]. The program does not directly communicate with hardware but via OS, unlike a DOS program. For example a DOS program can use functions such as "getchar()" to get keyboard input. In the event-driven programming environment the hardware, such as mouse and keyboard, are shared resources, and depending on the GUI state the keyboard input may be

directed to one or more windows for processing. One of the benefits of this approach is that multiple applications can be open at the same time and share the machine resources.

1.6 Macintosh Programming versus Windows Programming

Windows is based on the Macintosh. Both systems are quite similar. They are based on an event-driven GUI. User's interactions are through the standard interface elements, such as document windows, dialog windows, controls, menus, toolbar and so forth, although they are somewhat different on both platforms. Both systems use resources to describe user interface elements, and both systems present "similar" sets of APIs for managing the drawing of visual components, memory, and so forth.

On the other hand, The Macintosh is quite different to Windows in details, so porting an existing Macintosh program to Windows is not an automatic or trivial process. "The easiest programs to port will be those designed with a clear separation between the parts of the program that model the problem domain and the parts associated with the GUI"[B8]. For example, if the standard library functions in C++ are used in some code, this code is platform independent. However if member functions call OS APIs, the code has to be rewritten. The visual elements in an interface, such as windows, menus, push buttons, and dialogs, which are actually drawn and then displayed on the screen, are based on the OS APIs. This is why the interface for the PC version of the AudioGraph Recorder must be redeveloped.

1.6.1 Application Programming Interface (API)

The acronym API stands for Application Programming Interface. It serves as a software interface to be used by programs, and is truly the fundamental level of higher-level programming. It is a collection of functions or routines that are used to fulfil low level tasks, such as drawing on the monitor, disk access, printer interface, and Internet usage.

By using an API, programmers don't need to worry about the basic, fundamental chores involved in programs. For example, in Windows, the API Ellipse() function is called to draw an ellipse on the screen. The programmer doesn't need to define a structure for ellipse, and doesn't need to deal with each pixel drawing on screen. Another advantage is, for example, if new technology is used to make a more efficient way to render an image on screen, the corresponding programs do not need to be modified because the OS can be updated to include this new technology. Thus, using an API the program becomes device independent.

Different platforms have completely different APIs, which consists of hundreds API functions that are organised in a random way. "The API of an operating system is the fundamental reason why different operating systems are incompatible - why, for example, a Macintosh-based piece of software cannot run on a system running Windows 98 (without an emulator, at least)" [B9].

Some people may ask why Java is cross-platform? The reason is that Java provides a "virtual machine" mechanism that has the similar functionality of an emulator. An emulator acts as a sort of translator between the two APIs in two platforms, whose core function is "converting" the API calls of one program into the API calls of the operating system running it.

1.6.2 Operation system

Current Windows and Macintosh OS both are 32-bit, multithreaded and multitasked operating systems, and with similar user interfaces. They are developed by different companies and use different processors. They are similar but different in thousands of details.

A computer desktop usually refers to the file and application management environment. In Mac, there is a menu bar on the top of the desktop. The menus are changed as different applications are opened. An application in Mac doesn't have a main frame. In Windows, there is a task bar on the bottom, which can be dragged to any side of the desktop, whereas the menu bar in the Mac OS can't be. In Windows an application has its own menu bar contained in the application main frame.

The Mac desktop icons are more "intelligent" or "object-oriented". For example, a Mac desktop icon with an italicised name is an alias for the real file or volume, just like a Windows shortcut. A difference exists however, in which if a user moves the target file to another directory on the Macintosh, the reference of the alias that points to the target object is automatically updated, so the user does not "lose" the document or application.

In Windows, the same icon is used for an application and its document type file. Three-character extensions are used to specify file types, such as "doc" for MS Word files or "exe" for applications. In Mac, an application and its document file are represented by different icons. "Macintosh file types are not indicated by the file name, but rather by an invisible four-character identifier such as 'TEXT' for text documents, 'PICT' for picture files, or 'APPL' for applications"[B10].

Event handling in the Mac OS is also different to that in Windows. In Windows, all visual interface elements are treated as windows, such as frames, icons, menus and scroll bars. Windows receives all inputs and puts in its system queue. To redirect the input to an application queue is the task of the system queue. When the application is not idle, it retrieves messages in the application queue and dispatches messages to the target windows. In the Mac OS, icons, menus, and scroll bars are not treated as windows. "Events must be retrieved from the OS event queue by an application, and it is the application's job to handle the events and to request behaviour from windows, scroll bars, icons, etc. as necessary. While there are around 400 different message types defined in the Windows APIs, there are only 16 types of events defined in the Mac Toolbox. Conceptually, the Windows OS 'pushes' events to each window's message queue, while a Mac application's event loop must 'pull' events from the operating system"[B11].

Many operating systems, including Windows, treat files as ordered sequences of bytes. In Mac files, data and resource are stored in two separate parts of a file: the data fork and the resource fork[B12], as in Figure 1-1.

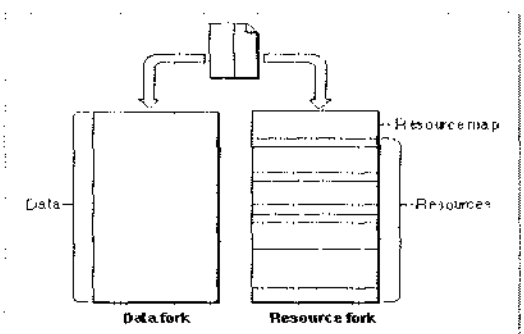


Figure 1-1. The Mac file

A file's data fork contains the file's data, which is an ordered sequence of bytes. A file's resource fork contains that file's resources, such as icons, menu resources, etc. used in an application, and preference settings, fonts, and location of windows for a document. Windows uses different files for the file's resource, for example storing preference settings in an associated ".ini" file.

For documents, both Windows and Macintosh support the Single Document Interface (SDI) and the Multiple Document Interface (MDI). However, in Macintosh, an application can't be launched two times at the same time, whereas in Window an application can be launched multiple times.

1.7 Tool for the development of the PC version AudioGraph recorder

For Windows, there are many tools for application development, such as Visual Basic, Delphi, Visual C++, and Java. They can all handle interface design pretty easily in some way. However, interface implementation is only part of a project. The core of an application is to provide certain functionality to end users to perform certain tasks.

To develop an AudioGraph Recorder for Windows, the correct selection of the proper tool will greatly reduce the development period, and improve the exiting application performance. Particularly, taking advantage of some platform-independent code in the exiting Macintosh version AudioGraph Recorder, which can be ported easily. So what are the important factors needed to make the decision? The key is the domain of this project. The AudioGraph Recorder is used to produce files that contain a set of data and objects, which is a "document-based" application.

1.7.1 Document-Based Application

"A document-based application provides a framework for generating identically contained but uniquely composed sets of data that can be stored in files"[B13].

Microsoft Word is one well-known example of document-based applications. The following lists common tasks that such an application can perform[B13]:

- Creating new documents.
- Opening existing documents that are stored in files.
- Saving documents under user-designated names and locations.
- Reverting to saved documents.
- Closing documents (if a document has been modified, give a prompt to users).
- Printing documents and allowing the page layout to be modified.
- Representing data of different types internally.
- Monitoring and setting the document's edited status and validates menu items.
- Managing document windows, including setting the window titles
- Handling application and window delegation methods (such as when the application terminates)

For document-based applications, handling text-type data is very easy, but handling a collection of objects, such as different graphics and sound, is more difficult. This may require different data structures and must deal with storing and retrieving objects to and from disks. Some architecture is invented to make programmers deal with this problem easily, such as the serialisation architecture in Microsoft Foundation Class (MFC) and Java. When using Visual Basic or Delphi, programmers must handle all this by hand. This one reason is enough to discard Visual Basic or Delphi from the list of the PC tool candidates.

For selecting a suitable tool for the project development, we will next compare the two programming languages: C++ and Java, and their corresponding frameworks: MFC and JFC.

1.7.2 C++ versus Java

C++ and Java are both popular and successful Object-Oriented languages. C++ "adds classes to the C language", so it inherits all the flexible and efficient features from C language. Java has gained experience from C++, which is similar in syntax but

completely different in internal from C++. The list below compares some different features between Java and C++:

- Pre-processor

  The C++ pre-processor performs an intelligent search and replace on the identifiers that have been declare by the #define or #typedef directives. Java uses declaring constant and class as to achieve the same effect and to keep program more readable.

- Structure or Unions, functions and Goto Statement

  Java has none of these features in order to keep the program 100% object-Oriented. However, the features are options in C++.

- Inheritance

  C++ uses multi-inheritance for class hierarchy, while Java uses single-inheritance. Multi-inheritance adds complexity in programs, but it reduces the code redundancy. For example, in a MFC single-inheritance class hierarchy, CButton class never uses the serialisation functions in the root class (CObject).

- Operator Overloading

  Java does not support operator overloading; instead it achieves this goal by declaring a class, appropriate variables, and appropriate methods for manipulating those variables. There is no doubt that operator overloading in C++ is very powerful and convenient feature.

- Automatic Coercion

  Java prohibits the C++ style automatic coercion between different types. It uses type casting instead.

- Pointers

  Java uses the pass by reference mechanism to achieve the same effects as using pointer in C++, which prevents bugs caused by the mismanagement of pointer. But using pointer is much more efficient as the application can directly access data in memory by its address.

Java is easier to use and to control error than C++. C++ is considered to be the language of choice for the professional programmer. Another advantage of Java is

portable, which means that a Java program is platform-independent. However, there are some drawbacks for Java, the main areas being:

- Although Java can be interpreted and compiled, Java programs are much slower than the program written in C++.
- "Java will not run on all microprocessors, only those microprocessors for which Sun Microsystems, Ins. has translated the Java language" [A4].
- As Java forces that programs use 100% pure Object-Oriented techniques, and the complex translation process of its virtual machine due to the purpose of platform-independent programming, the "The more complex a Java application (e.g. multimedia and multithreads applications), the slower its run-time performance" [A4].
- Java throws the learning curve to the programmers experienced in C, C++, Pacal etc, as Java is similar in syntax to those languages but it behaves differently.

1.7.3 MFC versus JFC

MFC is a complete application framework, which is specifically tailored for creating applications for Microsoft's Windows operating system [A6]. By using C++ and Object-Oriented techniques, it encapsulates most of the Windows API to simplify Windows development. In MFC, the classes can be grouped into four general categories:

- General –purpose classes

  These classes, such as string-handling class, list and map class so on, help glue the framework together.
- Windows API classes

  The Windows API classes wrap up the Windows API to encapsulate Windows objects, such as window class, dialog box class, control class and device context classes so on. The device context classes are used to present information to allow Windows to translate output request, such as drawing on the screen.

- Application framework classes

  These classes handle the main parts of the framework, such as the message-pumping logic, document-view architecture and printing etc. The document-view architecture is one of the cornerstones of MFC. It separates the actual data from the representation of that data by document class and view class. The document-view architecture helps glue documents, and the document-associated views together. A view is contained in a frame window.
- High-level abstractions

  These classes provide the advance features of windows, such as splitwindow, toolbar etc.

Layered on the top of the Abstract Windows Toolkit (AWT), the Java foundation classes (JFC) are a set of pre built classes, which provide Java applications and applets with the building blocks for a sophisticated graphical user interface. JFC is composed of five APIs:

- AWT is a platform-independent interface that allows development of a GUI that runs on all major platforms.
- Java™ 2D: "The Java 2D API provides a subclass of the AWT Graphics class, Graphics2D, with additional features for specifying fancy paint styles, defining complex shapes, and controlling the rendering process. The Java 2D API treats paths, text, and images uniformly; they can all be rotated, scaled, skewed, and composited using the new Graphics2D class"[B14].
- Drag and Drop: The generic, platform-independent implementation of D&D will enable objects to be dragged and dropped from one application to another.
- Swing includes a component set that is targeted at forms-based applications. Swing components have had the most immediate impact on Java development. They provide a set of well-groomed widgets and a framework to specify how GUIs are visually presented, independent of platform.

Compared to MFC, JFC is much newer, and is continuing to be developed. The major advantage of JFC is its "Write Once, Run Anywhere" philosophy. The design goal for MFC and JFC is different. The MFC was designed to provide the users with a

complete application framework, which would greatly simplify Windows programming and reduce the development period. "The JFC is the result of developer and end user feedback and provides a significant advance in the development of client applications written in the Java programming language"[B14], which previously aimed at the intranet and Internet applets and applications.

In the field of the basic user interface components and functionality, both MFC and JFC do a fairly good job. Both of them support the basic GUI components such as radio buttons, check boxes, label controls, combo boxes, list boxes, text controls, button, and tab control etc. They both also support menus, tool tips, scrolling window, tool bars, progress bars, tree controls. While both of them provide pre-built dialogs for opening files, selecting colours, MFC also provides dialogs for choosing fonts, and page set-up etc. In general, for the class hierarchy of the interface components, MFC is more complete than JFC.

For the event handling, JFC is completely different from MFC. JFC provides an event model called Delegation Model, which is based on two basic components:

- An event source, such as components, the keyboard and the mouse.
- An EventListener, which are defined for each generic class of events.

Let's look at an example of how buttons (JButton) handle mouse clicks. To detect when the user clicks a button, a program must have an object that implements the ActionListener interface. The program must register this object as an action listener on the button (the event source), using the addActionListener method. When the user clicks the button, the button fires an action event. This results in the invocation of the action listener's action performed method.

MFC uses a Message-Mapping mechanism to handle events. A message map for a class is simply a table of member functions that handle Windows messages. Each entry in the message map associates a function with a particular message. A message map for a class is created automatically in the Visual C++ environment. For the message handler definition, the macro DECLARE_MESSAGE_MAP() must be

included in a class definition, and the class implementation must include the macro BEGIN_MESSAGE_ MAP() end END_MESSAGE_MAP().

The event model in JFC requires implementing a listener class if an event is notified. Implementing a listener class is not only inconvenient, but also inefficient[B22].

MFC is a more complete application framework than JFC in the field of document-based application. The following is an example about a Multiple Document Interface (MDI) application.

In JFC, the child frame for a MDI application uses JInternalFrame class. A content pane must be added to the JInternalFrame class, which is used to render data. For the data shown in each child frame, it needs a document class to hold the data for storing and retrieving the data. Some structure must be set up for keeping the multiple documents and the relationships with the child frames. All this must be done by hand.

In MFC creating a MDI is very simple as MFC have a pre-built document/ view architecture to support MDI. The code used for creating MDI is simple:

```
pDocTemplate = new CMultiDocTemplate(
                    ID,
                    RUNTIME_CLASS(CmyDocument),
                    RUNTIME_CLASS(CmyChildFrame),
                    RUNTIME_CLASS(CmyView) );
```

By these codes, multiple child frames, views, and documents are automatically created and bound together. Behind these codes, MFC has done lots of work. The problem for such a code is difficult to understand until understanding the internal mechanism of the architecture.

Another advantage of using MFC is that the current environment (Visual Studio 6.0) for MFC provides the tools to help programming. For example:

- AppWizard for creating a skeleton of an application.

- ClassWizard for extending and customising the classes in an application.
- Resource Editor for creating or modifying such things as menu, toolbar and dialogs during design time

These tools provide great convenience to programmers. For JFC, in the current environment such as Code Warrior and Visual Studio 6.0 for J++1.1, all code must be coded by hand.

Because the AudioGraph Recorder is a multimedia application, the performance of the application is one of the key factors; and the current personal computers mostly use Windows or Mac OS; And if using C++ as the development language, it makes porting easier as the Mac AudioGraph Recorder was implemented by C++. As a result, MFC is chosen as the tool for developing the PC AudioGraph Recorder.

1.8 The difficulties of the project development

The first difficulty encountered in the project development was that the learning curve for MFC is long because MFC is an application framework. An application framework is "an integrated collection of Object-Oriented software components that offers all that is needed for a generic application"[A5]. The application framework is a superset of a class library. The classes in the application framework are more integrated and more interdependent than in a class library, which makes an application framework more difficult to use.

Secondly, the components of the Macintosh AudioGraph Recorder interface all can be designed in design time by using the resource editor in CodeWarrior, even some initialization of components such as font. However, by using MFC in Visual C++, only the dialogs, toolbars and menus can be designed during design time, other components such as frame and view must be created by code. Also when porting an application from Mac to PC, the first requirement is the consistency of the interface. A problem arises because the classes in MFC do not support some of the features of the components in the Macintosh AudioGraph Recorder interface. For example, to model the Tool Window that has a group of image buttons with two different sizes,

the CToolBar class was considered first, but CToolBar only support all the buttons in a toolbar with the same size, so the CDialogBar class is used. CDialogBar however fails to support image buttons, so a new class has to be derived from DialogBar class and some additional functionality needs to be performed for the new class.

Summary

This chapter discussed some issues about the background of the AudioGraph authoring tool, the user graphic interface, the application domain (document-based), and the tool (MFC) for the development of the PC version application. Some of these issues are discussed in more detail in later chapters.

This thesis is organised as:

- Chapter 2 – discussing the GUI difference between Windows and Macintosh. And analysing the interface of the Macintosh version AudioGraph Recorder.
- Chapter 3 – discussing the framework difference between MFC and PowerPlant. And analysing the portability of the Macintosh version application.
- Chapter 4 – Comparing a SDK program with a MFC program and discussing some internals of MFC.
- Chapter 5—Explaining the detail about the interface design and implementation of the PC version AudioGraph Recorder.
- Chapter 6– discussing the achievement and future development for the project.

Chapter 2 The analysis of the Macintosh AudioGraph Recorder interface

This project concerns the porting of the AudioGraph Recorder interface from the Macintosh platform to the PC platform. This chapter is primarily concerned with the introduction to the interface and the functionality of the AudioGraph recorder on the Macintosh, especially how users interact with the interface and the state changes of the interface controls. The existing application therefore will act as a specification for the new interface on PC.

The AudioGraph Recorder is one of the tools of the AudioGraph authoring system, which is based on the model of the lecture theatre or presentation room, where a presenter arrives with a set of slides, which are presented, highlighted, animated with colour pens and finally talked to. It is used to record a presentation for subsequent playback on the Web.

As in the lecture theatre, to record a presentation, users first need to create a series of slides using a presentation graphics package such as Microsoft PowerPoint. The slides can then be imported as presentation graphics by the AudioGraph Recorder. "Alternatively the slides can be graphical objects cut from any application running under Mac OS or 'printed' from any application using the Macintosh Print-2-PICT printer driver. The PICT image format is the standard format used on the Macintosh and is used by the AudioGraph, the clipboard and many other graphical utilities"[B5]. The slides are displayed on the presenter's screen in the AudioGraph package, as graphics objects or components.

In the lecture theatre, the presenter will show slides one by one to students, and simultaneously the presenter may speak, and highlight or annotate the presentation material with tools, such as pointing device and coloured transparent pens. In the AudioGraph lecture recorder, this modelling is provided by a complete set of pointing

and drawing tools for annotating the slides and for drawing attention to components of the slide being presented. "The final and most important component provided in the AudioGraph lecture recorder is the ability to record voice annotations on the slide, and to sequence voice and other annotations"[B6]. The AudioGraph Recorder also provides rich utilities for editing annotations, such as users being able to change the colour for highlighting annotations and drawing annotations, recording sound and importing a new graphic etc.

After finishing a presentation using the AudioGraph Recorder, the user can save the document into a **.aep** file or a **.html** file. The **.html** file just embeds the corresponding **.aep** file with it. Both files are used to play back the presentation in Netscape or Internet Explorer browsers with the Audiograph plug-in. The standalone Player can also playback the **.aep** file that is stored on the local machine or CD ROMs.

As the AudioGraph Recorder adopts the conventional Macintosh user interface form, which has typical components of a modern GUI, it very easy for users to use and interact with.

Part 1 The graphics user interfaces of Macintosh and Windows

2.1 GUIs for Macintosh and Windows

Macintosh and Windows have very similar GUIs that perform similar functionality. As an example of a multiple document interface (MDI), Figure 2.1 shows the GUI differences between Macintosh and Windows.

In the menu, differences are that the Macintosh keyboard accelerators are triggered by command (Apple) key (such as Apple+C), but Ctrl or Alt key triggers the Windows keyboard accelerators (such as Ctrl+C). On the File menu, Windows supports the Print Preview feature, but the Macintosh.

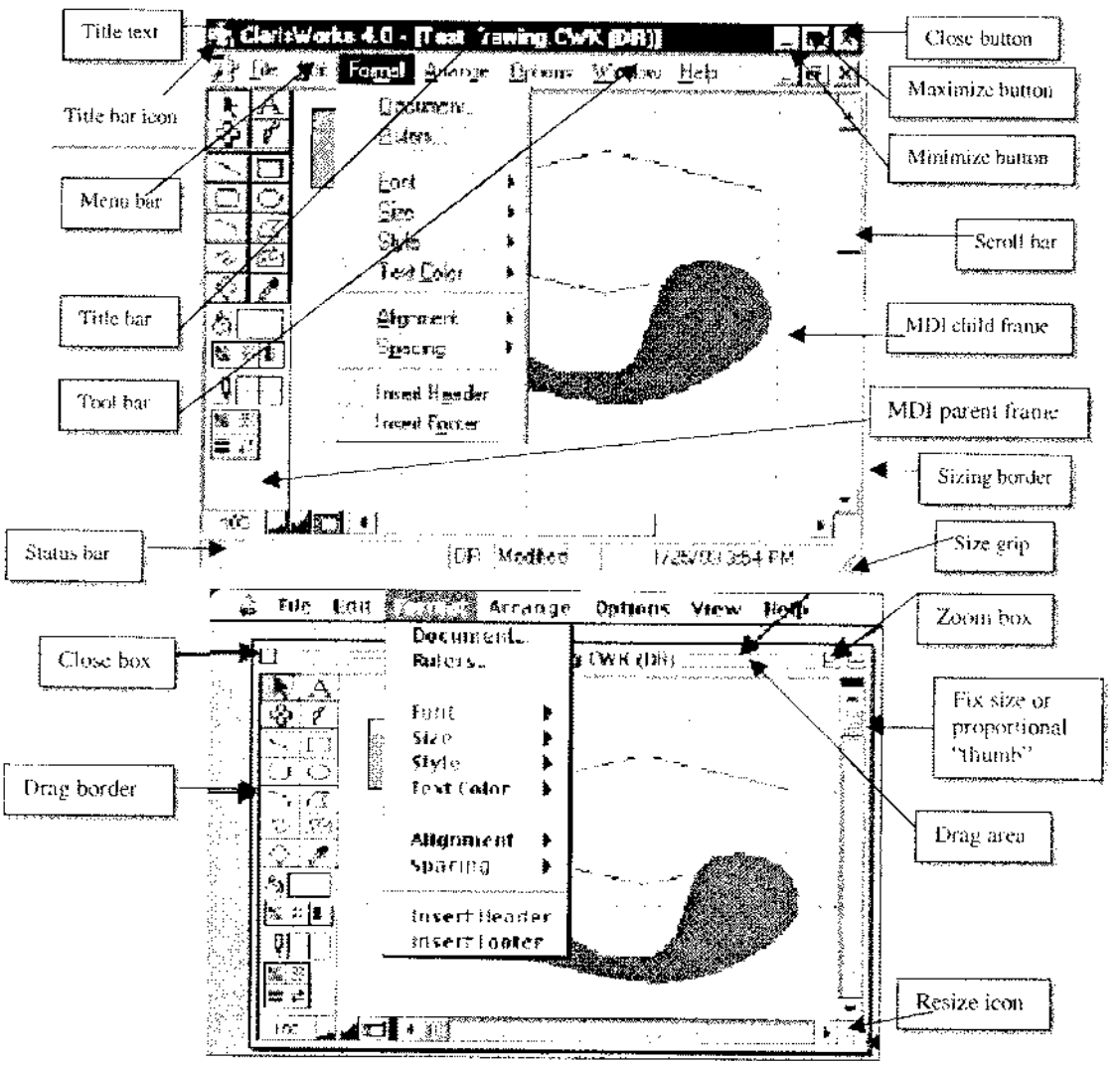


Figure 2-1 Windows 95 vs. Mac OS 8.5 window

For frames such as the MDI frame, Mac does not support the sizing border (resize a window by dragging its border). A frame can be resized only by the resize icon. A frame in Windows however can be resized by dragging either the frame border or the size grip.

The Macintosh tool bars tend to be left floating until the user is finished with them—then they are closed, as Macintosh application does not have a main frame. By comparison, Windows supports not only "floating" but also "docking" toolbars, which means that tool bars can "dock" on any side of a frame client area into a fix tool bar.

In the Mac OS, the horizontal and vertical scroll bars for a scrolling view are never hidden, whereas in Windows, both scroll bars do not appear until the contents that appear in a view are larger than the current view port size.

2.2 The states of GUI components

With regard to the common components (including frame, controls and dialog etc.) in GUI, there are the following defined states:

- Visible/invisible state, which controls whether a user can see the component.
- Enabled/disabled state, which determines whether a component responds to a mouse-click. This state affects the appearance of a window, for example a button in the disable state is dimmed.
- Active/inactive states, which means whether a component is in the foreground or background of the desktop. For example, if a window in Windows is clicked, the window become active with the background colour of the window title bar changed into deep blue. In GUI, usually only one window is active at one time.

Some controls in a GUI also have their own status, such as checked/unchecked for radio button and check box, and normal/pressed (or checked) for push button.

2.3 The development process of GUI

The development of GUI usually includes four stages or phases:

- Planning
- Analysis
- Design
- Implementation

Each phase is not completely separated from the others, as often refinements need to be made. In other words, there may be iteration and concurrency between the phases. There are some process models that are adopted by developers for the development of software. One of them is the Waterfall model as shown in Figure 2-2.

In the Waterfall model, the processes flow in one direction. The assumption is that the deliverables of each phase meet their criteria and the knowledge for the project (e.g. specifications and technology) will not change over the lifetime of the project. It lacks iteration and overlap between phases.

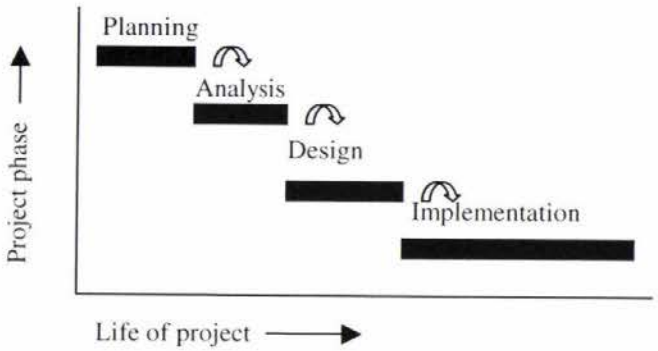


Figure 2-2. Waterfall model of software development

This is usually not viable, as most environments keep changing. Data collection is difficult and changes very often, and updates to the software happen all of the time. Therefore the Waterfall model is risky. One of the currently poplar models is the Spiral Process model [C10] as shown in Figure 2-3, which uses a "spiral process" to manage the risks associated with the Water Fall model.

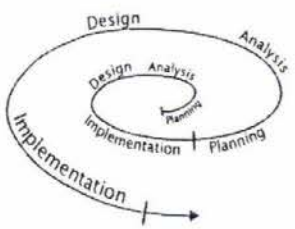


Figure 2-3. Spiral model

In particular, spiral development is synergistic with GUI design activities. In any case, the user interface design can be refined in each iteration, parallel with other activities.

In the spiral model, instead of delivering a complete product, a partial product is delivered to end-users over time. The problem with the model that it is difficult to manage the overlap phases and end-users may resist any changes made during the iterations.

For this project development, the spiral model will be used. This is because, although the source codes for the Macintosh version application are available, there is little documentation on its implementation available. Moreover, more features are being added to the Macintosh Recorder as it is still in development.

We will next discuss the interface of the AudioGraph Recorder.

Part 2 The interface of the AudioGraph Recorder

2.4 Starting point

Since the introduction of the "icon" concept that presents an object and brings it into the user interface, programming has had profound revolution when reviewing traditional command line programming. As an alternative to opening an application by typing a command into a command line, an application can be opened by double clicking the cursor on an application icon or on a document icon belonging to that application. The AudioGraph Recorder icon is shown in Figure 2-4.

Figure 2-4. The AudioGraph icon

When the AudioGraph Recorder is started, it will create an empty **AudioGraph Lecture Document.** The first start-up interface is shown in Figure 2-5:

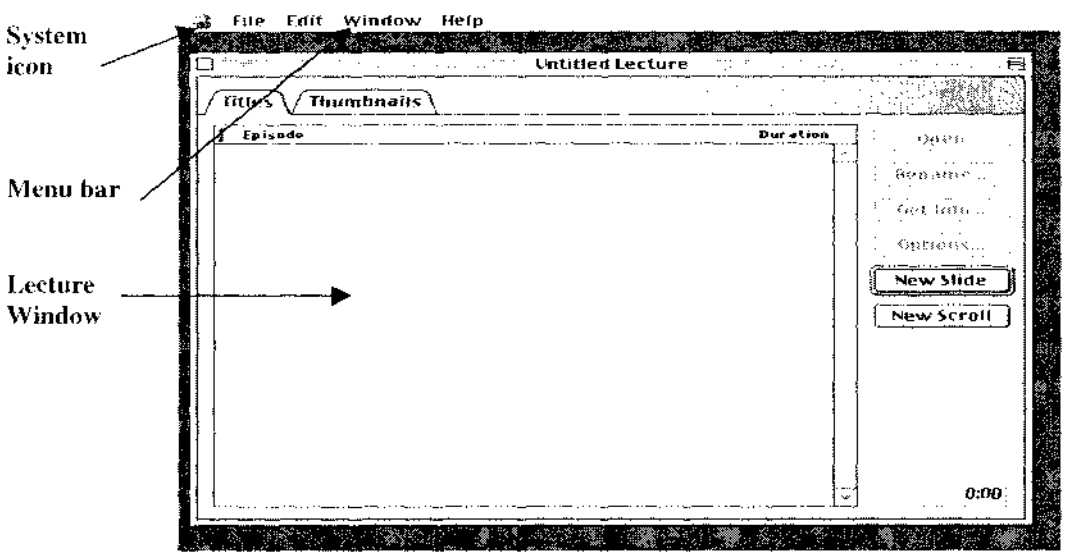


Figure 2-5. AudioGraph start-up interface

The interface consists of two components; the Menu Bar, and the Lecture Window that contains an empty **AudioGraph Lecture Document** at the present time.

As mentioned in the chapter one, the interface is slightly different with applications in Windows. In Windows, every application has its own application frame or window, which is the only parent window to its Menu Bar, Toolbar or other windows. However, on the Macintosh, the desktop is the application window. When changing to another application, the former one hides all its child windows in the background of the desktop.

2.4.1 The Menu Bar for the Lecture Window

"Menus, by definition, contain a list of items or names that represent options that the user can select"[A9]. When a menu item is selected, a message will be sent to the application by the operation system, which will cause some process occur, such as pasting some text.

According to the requirements during run time of the application, the menu items in the pop menu are dynamic, e.g. when a picture is selected, the menu item **Can't Cut** becomes **Cut Picture** etc. Figure 2-6 shows the menu bar in Figure 2-5 with all pop menu items.

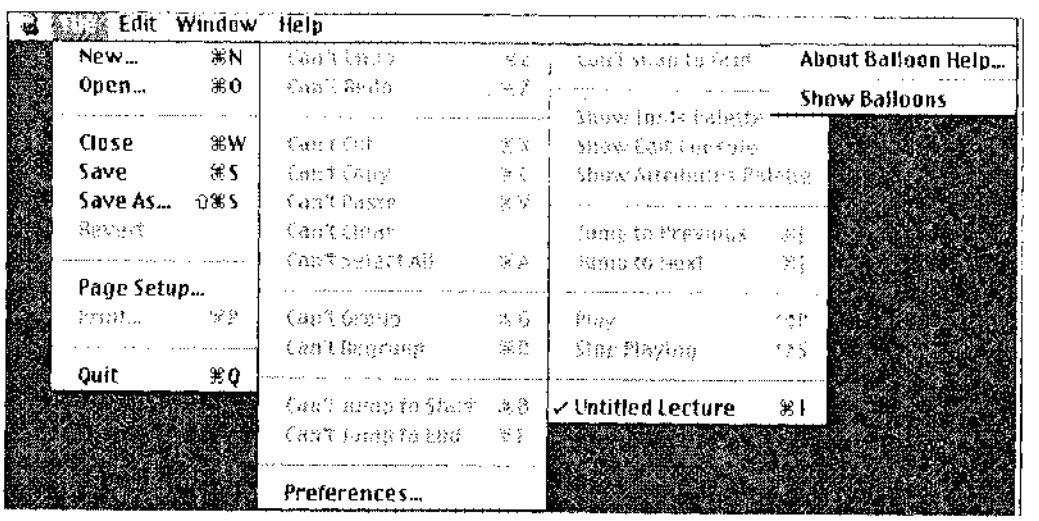


Figure 2-6. The AudioGraph Main Menu

In Figure 2-6, the screen shot combines all of the options of the four menu items in the main Menu Bar. The metaphor for menu items in the menu bar is actually button. When selecting a menu item by mouse clicking, or pressing a "hotkey", the appearance of the selected item button changes into the selected state (which can be seen as the **File** item selected in Figure 2-6), and a drop menu appears showing all the options. In the drop menu, some options are dimmed, which means they are currently in the disabled state that can't be operated. The symbol "**...**" behind the text of an option item means that a dialog box will be brought up for some parameter control if this item is clicked. Some options, such as "**New...**", are followed by keyboard accelerators. A separator in the form of a line in the drop menu is used to separate a group of options to make them easier to "see".

If no document is opened, meaning all lecture windows are closed, the main menu will change the option **New...** into **New Lecture** under the **File** menu. If the **New Lecture** option is selected, a new Lecture Window will open.

2.4.2 The Lecture Window

The Lecture Window in Figure 2-5 is used to manage the set of Slides or Scrolls within a single presentation or lecture, which is shown in the Figure 2-7 with the marks of all components and controls.

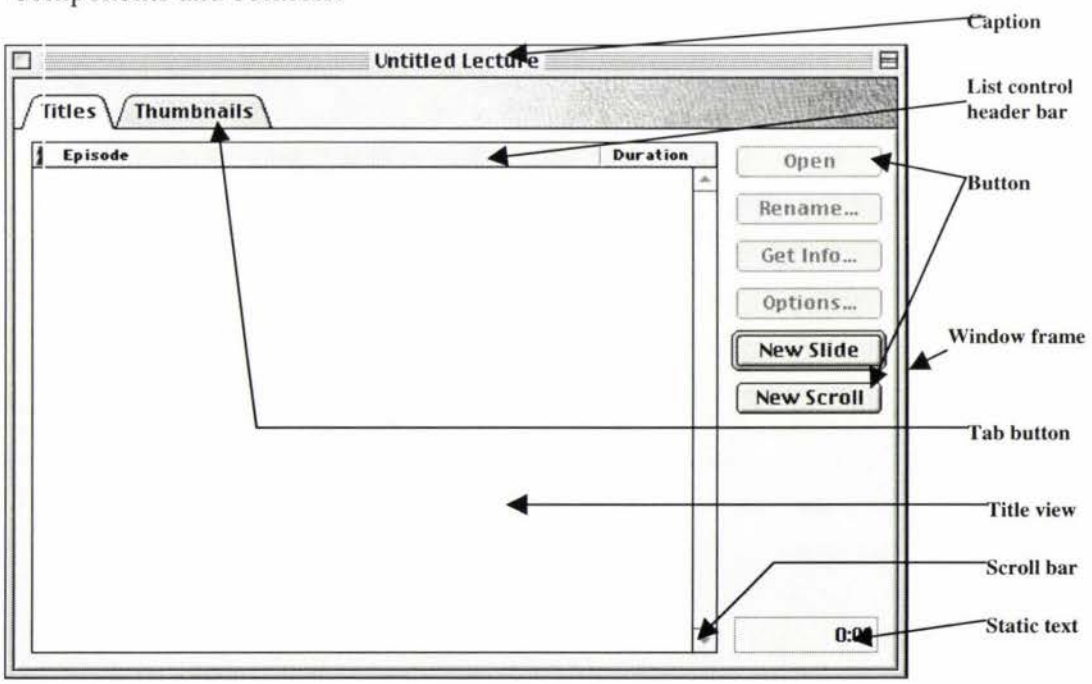


Figure 2.7. Lecture Window

The Lecture Window has a title bar with the caption "Untitled Lecture" initially, which will change to the file name when saved, a close box and a minimise box. It has a sunken client edge for its client area. There are some controls in the client area: six buttons on the right, a static text and a tab control. The tab control controls two views; the title view that can show a list of the slide titles and the duration for each slide, and the thumbnail view that can display the thumbnail pictures that represent the slides. The title view is the default.

Only two of the six buttons are enabled, the others are in the disabled state. The New Slide button is used to create a slide, and the New Scroll button is used to create a scroll. The difference between a slide and a scroll is that a slide is of a fixed size, whereas a scroll is of unlimited size in one dimension.

2.5 The creation of a lecture document

To create an annotated lecture, PowerPoint is first used to create a presentation and save it as a **Scrapbook (Pictures)** file. The file can be imported to the Recorder by click **Open...** under the **File** menu in the AudioGraph main menu. After choosing the file from the standard file choice dialog, the file created by PowerPoint will be displayed in the

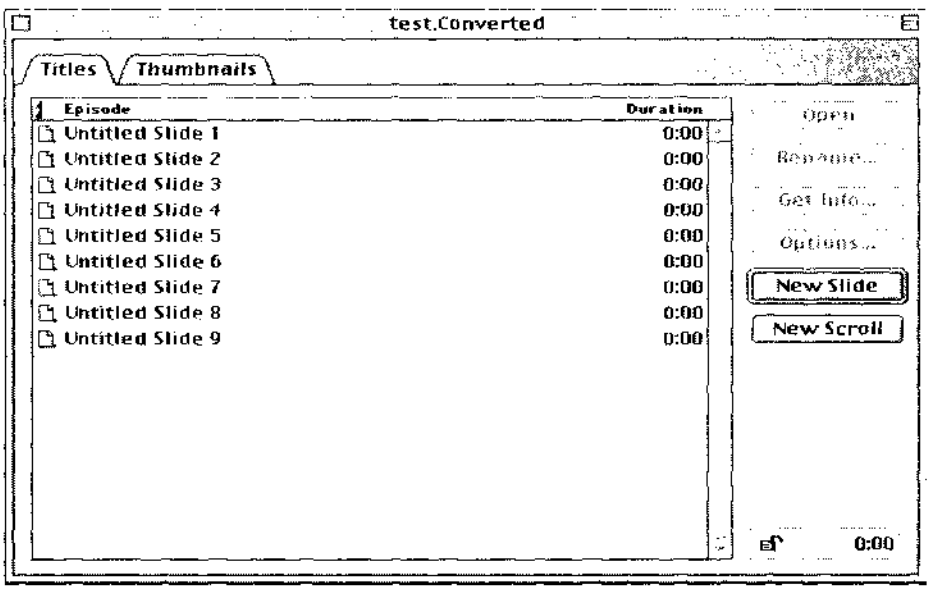


Figure 2-8. The Lecture Window with a document

lecture window. The Lecture Window now contains an **AudioGraph Lecture Document,** as shown in Figure 2-8.

The caption in the title bar is now changed, "test" as the name of the **Scrapbook (Pictures)** file**.** Several rows with two columns in each row and separated by lines are inserted into the title view. In each row, the first column starts with an icon to indicate if it is a slide or a scroll and if it is changed and not saved, followed by the title of a slide. This is not outputted by PowerPoint on the Macintosh. The second column indicates the time duration for the slide. On the bottom right of the Lecture Window, an icon showing whether it is read-only or not, is inserted in the left of the static text. The number following is the total time for the lecture. If there are too many slides, the scroll bar may be activated.

In the title view of the Lecture Window, each row can accept single-click and double-click events. Single clicking on any row will highlight the row and enable some buttons as illustrated in Figure 2-9.

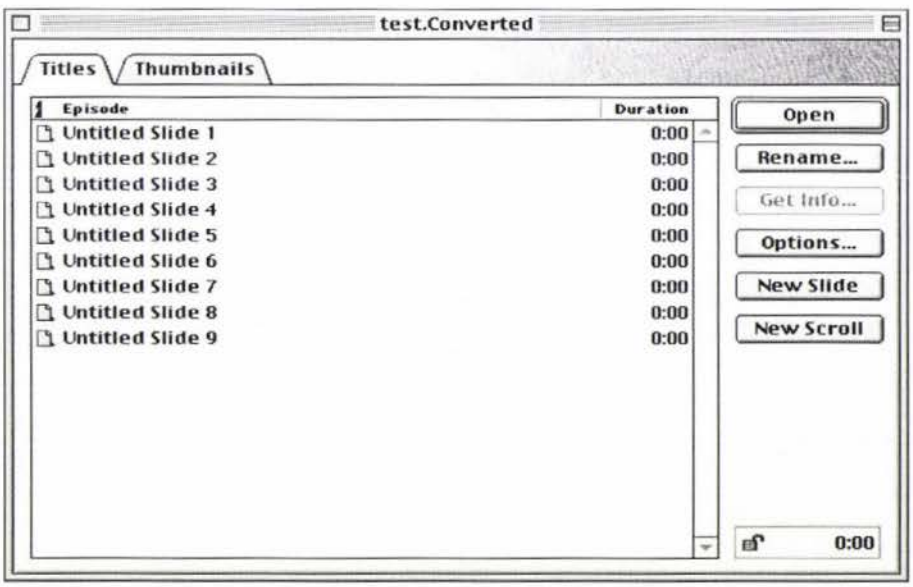


Figure 2-9. Highlights a slide in Lecture Window

The alternative view in the Lecture Window is the thumbnail view, which is obtained just by clicking the **Thumbnails** tab button, which is shown in Figure 2-10.

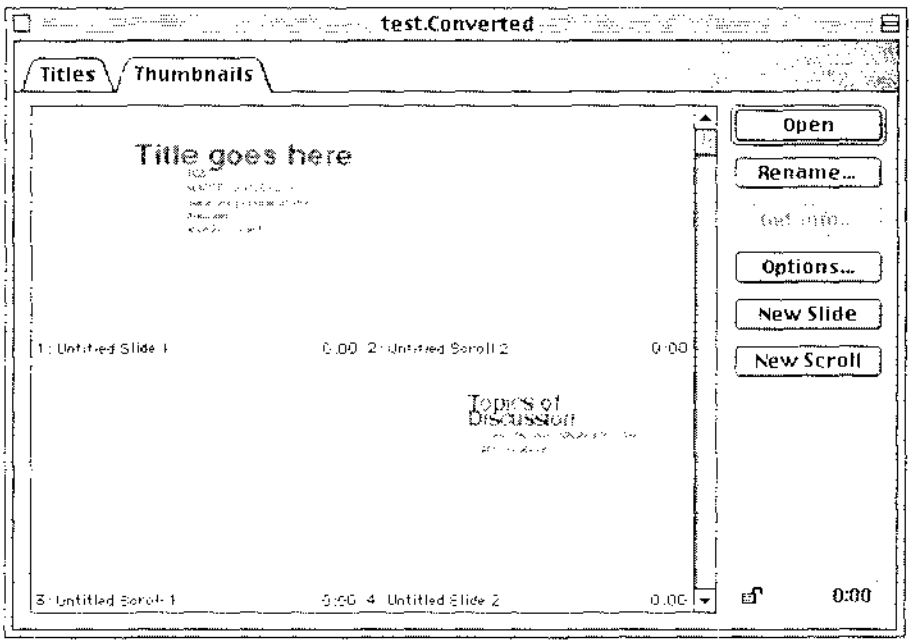


Figure 2-10. The thumbnail view of the Lecture Window

If the slide title or name needs to be changed, the use can click on the Rename... button in the Lecture Window. A dialog box such as in Figure 2-11 will show up.

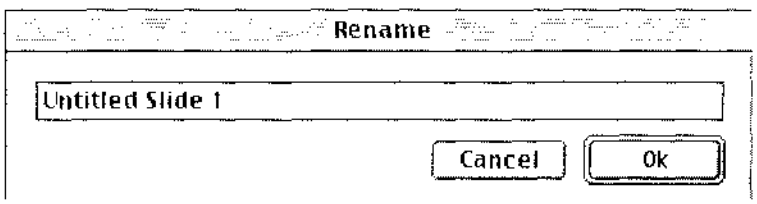


Figure 2-11. The Rename dialog box

The Rename dialog has a list box so that the slide name can be typed in to make the change. It also has a title bar and two buttons.

By clicking the **Option...** button in the Lecture Window, an information dialog appears as in Figure 2-12, which includes information for the highlighted slide or scroll, and which can be changed as the user likes. The metaphors used in the dialog include: text, edit box, group box, radio button, check box and button. The texts in the dialog use two types of fonts.

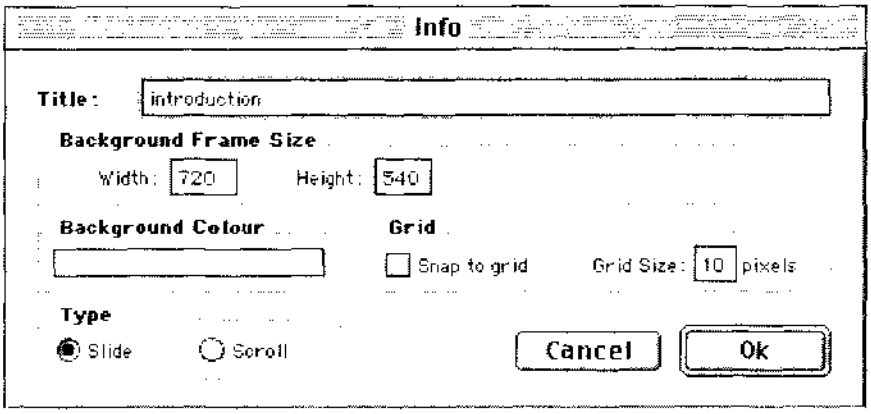


Figure 2-12. The slide information dialog

In Figure 2-9, when a slide is highlighted, a message is sent to the application to make some changes to the main menu. Four menu options under **Edit** menu item are changed and become enabled as shown in Figure 2-13. The **Revert** options under **File** and the ***Jump to* Next** *options under the* **Window** *item will also become enabled.*

So far we have seen a set of slides or scrolls created in a lecture using the AudioGraph Recorder from an existing file created by PowerPoint. Alternatively a slide or a scroll can be created by clicking the **New Slide** or **New scroll** button in the Lecture Window, the image added in the slide can be from either the clipboard or from a file.

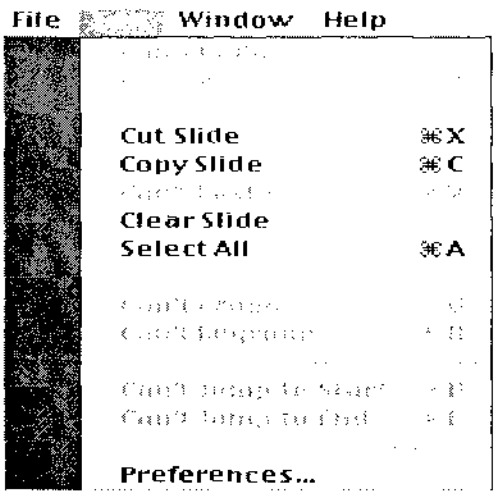


Figure 2-13. The menu when a slide highlight

Another way to create a slide or scroll is from the **File** menu to click the **New...** option, which will open a dialog box as in Figure 2-14, then select **Slide** or **Scroll**.

As a new slide is created, a new name will be automatically added to the title view and the

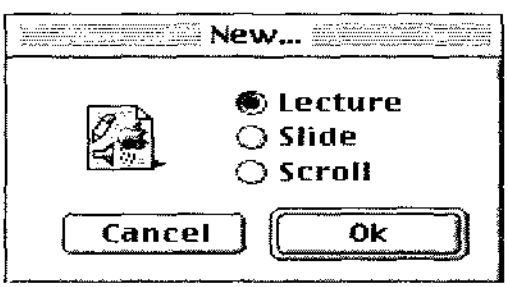


Figure 2-14. The selection dialog

thumbnail view.

If the Lecture radio box in Figure 2-12 is selected, a new Lecture Window will open, which means the application can have multiple documents open at the same time.

2.6 Annotating a slide

In the Lecture Window, if the user double clicks a slide title or highlights a slide first, then clicks on the Open button, a Slide Window with two tool windows will be opened as shown in Figure 2-15.

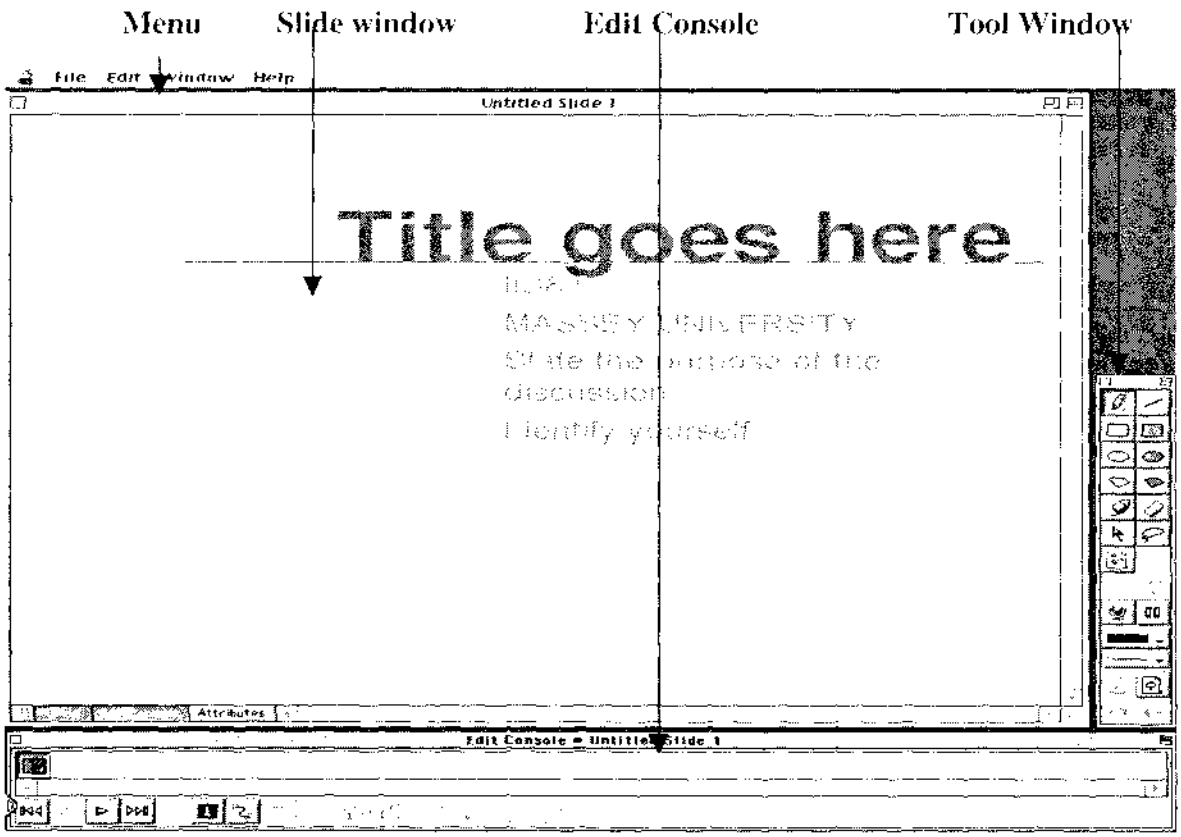


Figure 2-15. The interface with Slide Window

In this example, the Slide Window contains an image as its background in its client area, which is the working space. On the right hand side of the Slide Window, the Tool Window displays a collection of tools for annotating the slide. The Edit Console window

is below the Slide Window. "This window denotes the presentation from a temporal viewpoint, in which the objects that comprise the presentation are represented as icons. The sequence of icons represents the flow of time, with the first annotation on the left of the display and the last on the right. This example has just one annotation, an image, which is currently selected. If you select an icon in the edit console it is also selected in the slide window and visa versa and this gives a simple model for both spatial and temporal editing of presentations" [B5].

When an annotation task is made in the Slide window, a corresponding icon will be inserted into the Edit Console in sequence. The edit console can be used for playing the presentation or for displaying the presentation at a given point in time.

As the Slide Window appears, the Lecture Window becomes inactive and hides on the background of the screen. When the Slide Window is in inactive state, the tool window and the Edit Console become invisible. Also some menu options in the main menu are changed. Multiple Slide Windows can be open at the same time. If shifting from one Slide Window to another, the status of the Edit Console, Attributes window and Tool window will be changed and loaded according to the currently active Slide Window.

2.6.1 The Slide Window

The Slide Window has a common frame and a title bar. The frame is resizable using mouse dragging on the resize zone in the right bottom of the Slide Window. The title bar consists of the slide name as a caption in the middle, and a close box, a minimise box, and a restore box for restoring the former position and size of the Slide Window. A view completely covers the client area of the Slide Window. In modern GUI, a view is placed in a frame and is used to render data. For example, Figure 2-15 shows the Slide View containing a slide image at present.

The Slide Window view has a vertical scroll bar and a horizontal scroll bar, four button controls. The left-most button is the **Grid** button. With the **Grid** button checked, the

movement and positioning of graphical objects in the Slide Window is constrained to a grid which is set by the slide information dialog. The other three buttons (**Tool, Edit Console, Attributes**) control the visibility of the three tool windows: the Tool Window, the Edit Console window, and the Attributes window. The default state for the Tool button and Edit Console button is checked, and the Attributes button unchecked, so that the Attribute Window is not yet shown in Figure 2-15.

2.6.2 The Tool Window

The first time a Slide Window opens, the Tool Window in Figure 2-16 shows the original status with the Pen Drawing button checked and six buttons disable. It has a title bar with a close box and minimise box, a frame that can't be resized, and some images buttons in its client area.

The graphics tool that is selected, is indicated by both the cursor icon, which changes to indicate the tool selected and by the highlighting of that tool in the Tools window. The drawing tool buttons are grouped together, which means if a button is clicked, it becomes checked and the previous checked button in the group becomes unchecked.

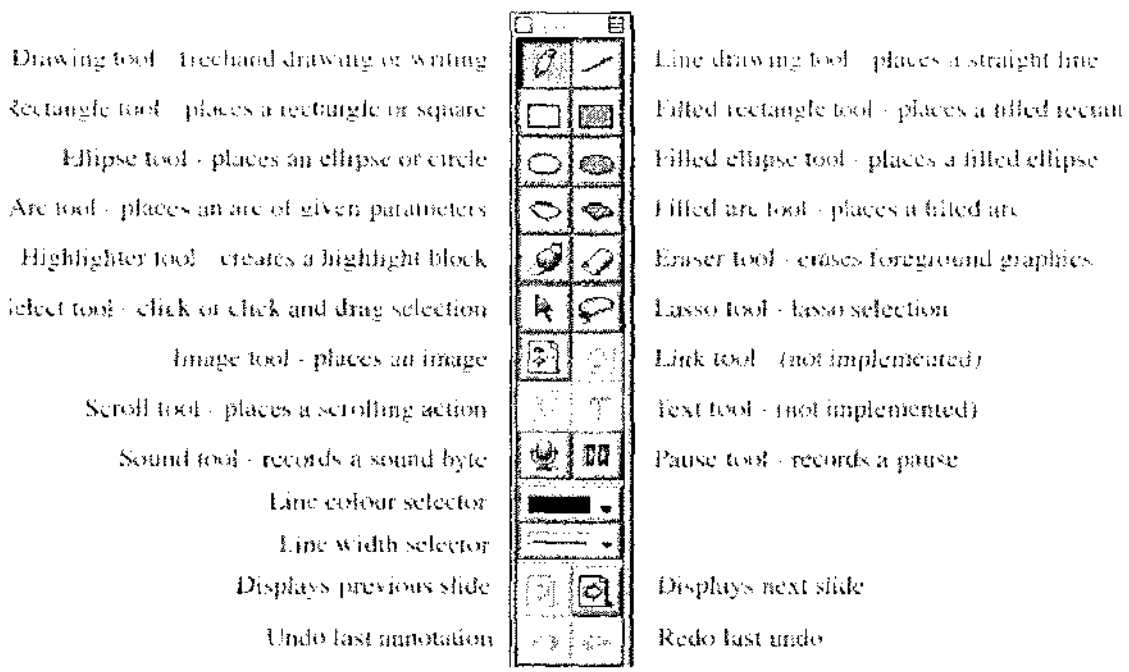


Figure 2-16. The Tool Window

An image can be put in the slide by clicking the Image Tool button. When the button is checked, the cursor becomes a small cross icon, then by holding and dragging the left button of the mouse, a rectangle picture frame can be defined in the slide window. On releasing the mouse, a dialog box appears for the choice of image files as in Figure 2-17.

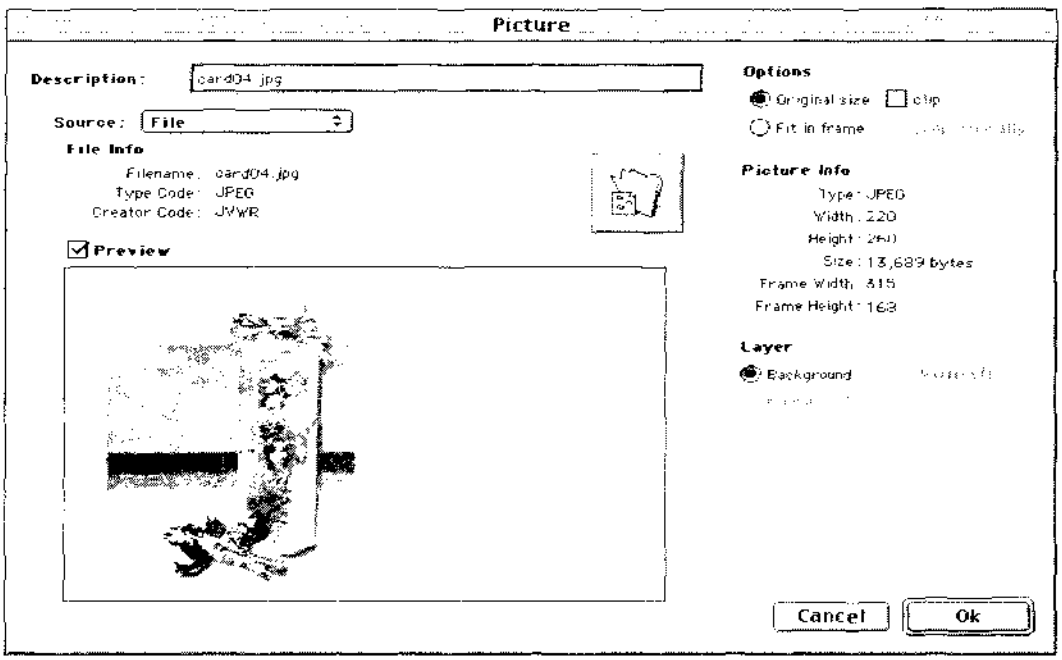


Figure 2-17. The Picture Choice Dialog

By clicking the open file icon in the dialog, a standard file choice dialog will show up. After the image file has been chosen, the preview will appear in the Picture Choice dialog. Pressing the OK button will cause the image to appear in the Slide Window.

When the Sound Tool button in the Tool Window clicked, a **Recording...** dialogue as in Figure 2-18 will open to begin the sound recording process. The other windows will be disabled. This allows up to one minute recording, which will be terminated by a second clicking on the Sound Tool button. There are two progress bar to indicate the memory usage and input gain, and a group of

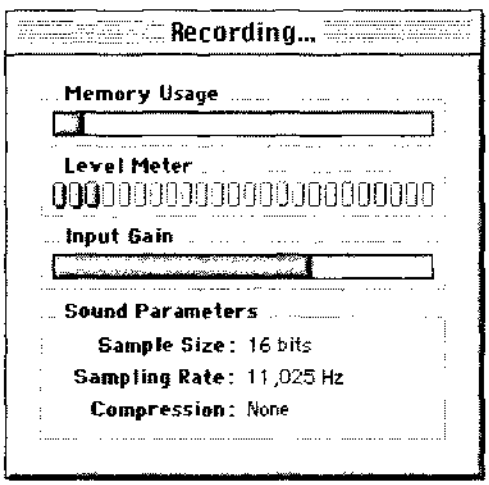


Figure 2-18. The Record Dialog

small buttons to show the level meter. If the level meter shows continuous excursions into the red region, sound peaks will be clipped and a distorted sound will result [B6].

When the Pause button in the Tool Window is clicked, a Pause dialogue as shown in Figure 2-19 will open to allow the pause time to be set. Pauses are used to control the speed of playback of non-real-time annotations or to insert pauses between a sequence of sound bytes.

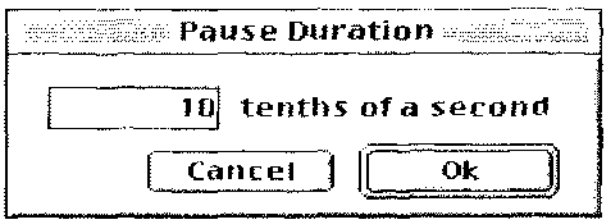


Figure 2-19. The Recording Pause Dialog

The colour selector button and the line width selector in the Tool Window have a larger size than the other buttons in the Tool window, with two icons in each button, one of them is the arrow icon. If they are clicked, drop-down menus appear as in Figure 2-20. If the user chooses an option from the drop-down menus, the icon on the corresponding button will change. In the pen width button's drop-down menu, the selection of the **Other...** option will bring the pen size dialogue shown in Figure 2-21 to allow the pen size in pixels be edited.

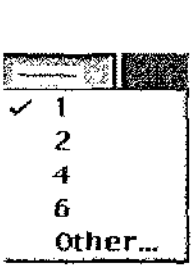


Figure 2-20. The drop-down menu

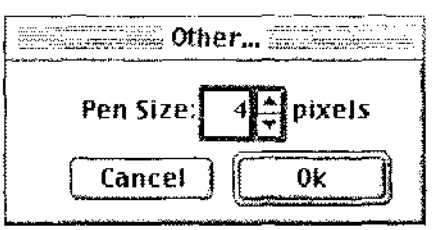


Figure 2-21. The pen size dialog

The Undo annotation button and the Redo annotation button in the tool Window are grouped together - just one is enabled at a time.

2.6.3 The Edit Console window

The Edit Console records the sequence of annotations that have been placed on a slide or scroll. When a tool button in the Tool Window is selected and an annotation is made in the Slide Window, a corresponding icon is inserted into the Edit Console window as shown in Figure 2-22. "Only voice and pauses have a real notion of time, the other annotations are graphical annotations and are just rendered in strict sequence"[B6].

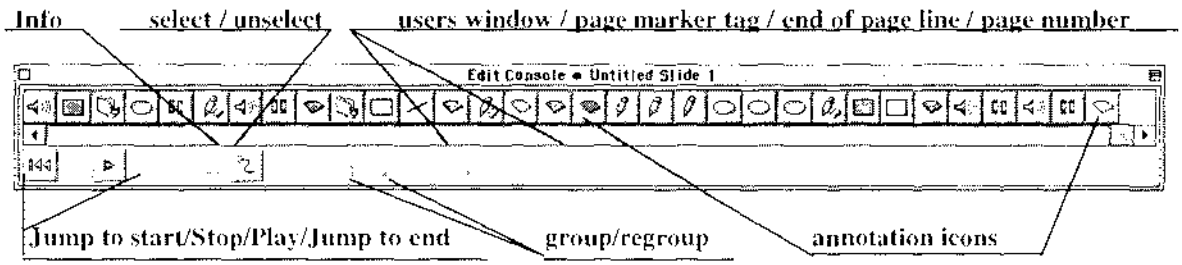


Figure 2-22. The Edit Console Window

The Edit Console window consists of a title bar, a frame that can't be resized, a toolbar with a set of editing buttons on the bottom, and a view with a horizontal scroll bar where a set of icons can be inserted.

If an annotation icon is checked, only those annotations up to and including that annotation are displayed in the Slide window. When the Play button is clicked, playback will start from the selected annotation.

When an annotation icon is selected, the Info button will be enabled. By clicking on the Info button or double clicking the annotation, a dialog for the corresponding annotation type will displayed where the property of that annotation can be edited. Some of the dialogs are the same as those displayed by selecting the tool button in the Tools Window. Figures 2-23 to 2-25 show different dialog windows.

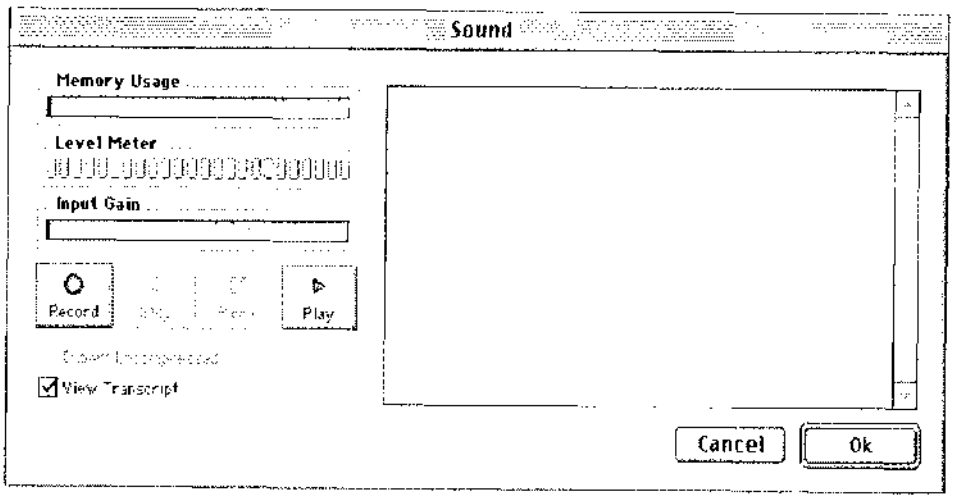


Figure 2-23. The sound Info Dialog

Note : If longer sound samples are required for some reason, the sound can be recorded within another application, such as Sound Hack for example, and copied and pasted into the AudioGraph Recorder[B6], or it can be broken down into a number of sound annotation.

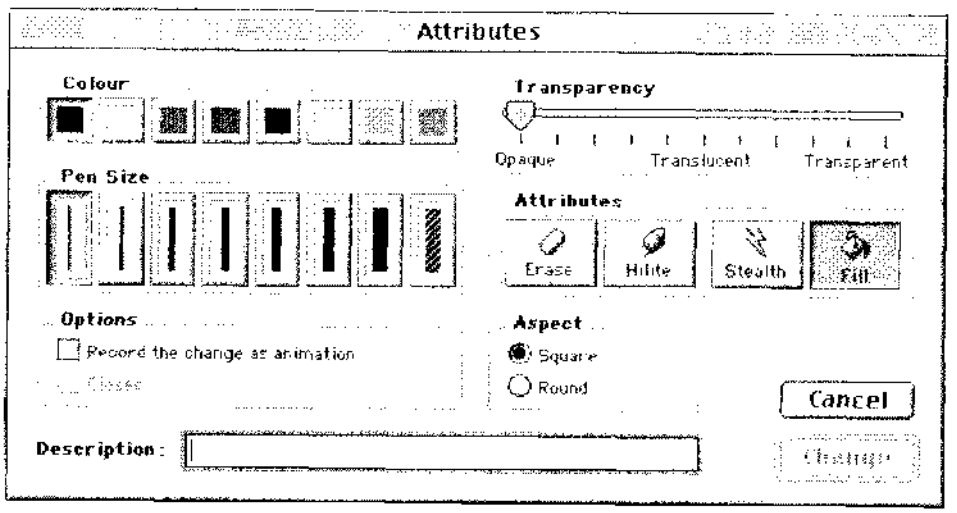


Figure 2-24. The Attribute/Info dialog

Note: The Attribute/ Info dialog is used for editing graphical annotations.

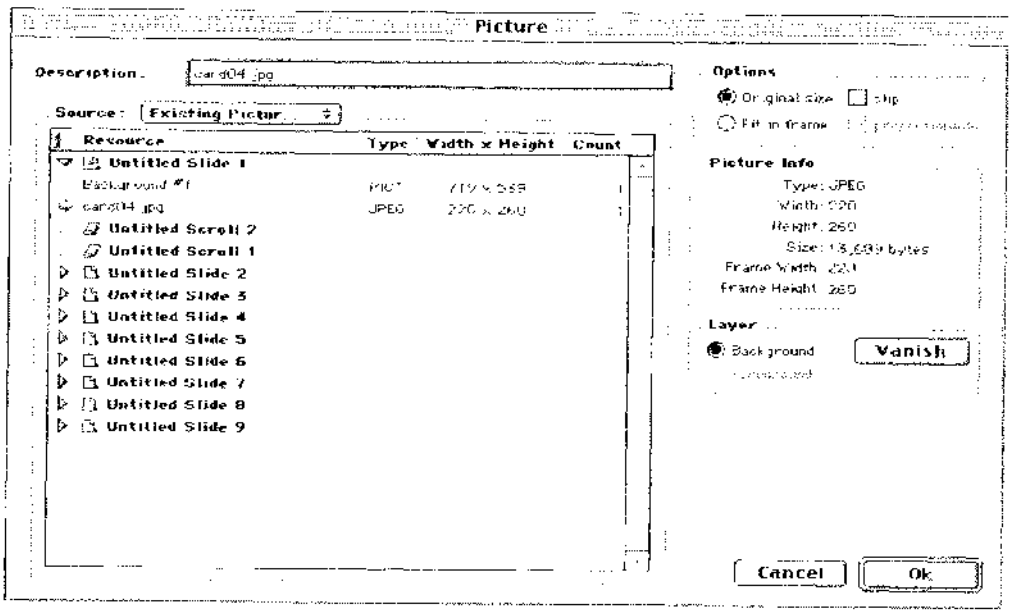


Figure 2-25. The Picture/Info dialog

Note: the Picture/Info dialog is for picture annotation editing. In the list view of the dialog, if the triangle icon on each row of the list is rotated by clicking it, it will show the picture files in the slide.

2.6.4 The Attribute Window

By clicking the Attributes button in a Slide Window, an Attribute Window is opened as shown in Figure 2-26.

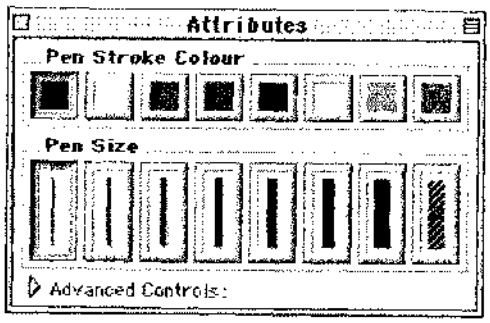


Figure 2-26. Attributes window

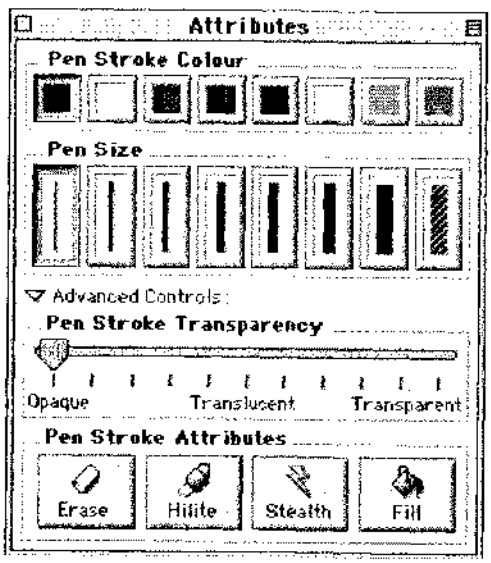


Figure 2-27 Attributes window extended

The Attribute window has two groups of image buttons for the basic and advanced attributes. A small triangle icon on the window is used for extending the dialog as shown in Figure 2-27.

Colour, **Linewidth** and **Transparency** values can be set in this dialogue, when a foreground graphical annotation is selected. Rectangles, ellipses and arcs can also be filled using the fill attribute. The **Stealth** attribute is a means of modifying the normal rendering action. It allows groups of annotations to be rendered together and then all displayed at once [B6].

2.7 The reference dialogue

The preference dialog is opened by selecting **Preferences...** from the **Edit** item in the menu bar. It sets up a number of user defined settings which are stored in a preferences file and loaded each time the program is started. A tab control or page control with four views or pages is shown in the dialog. They are; Sound, Graphics, Editing and General views. The **Factory Settings** button reverts to a default set of preferences. The following Figure 2-28 only shows the dialogue with the sound preference page.

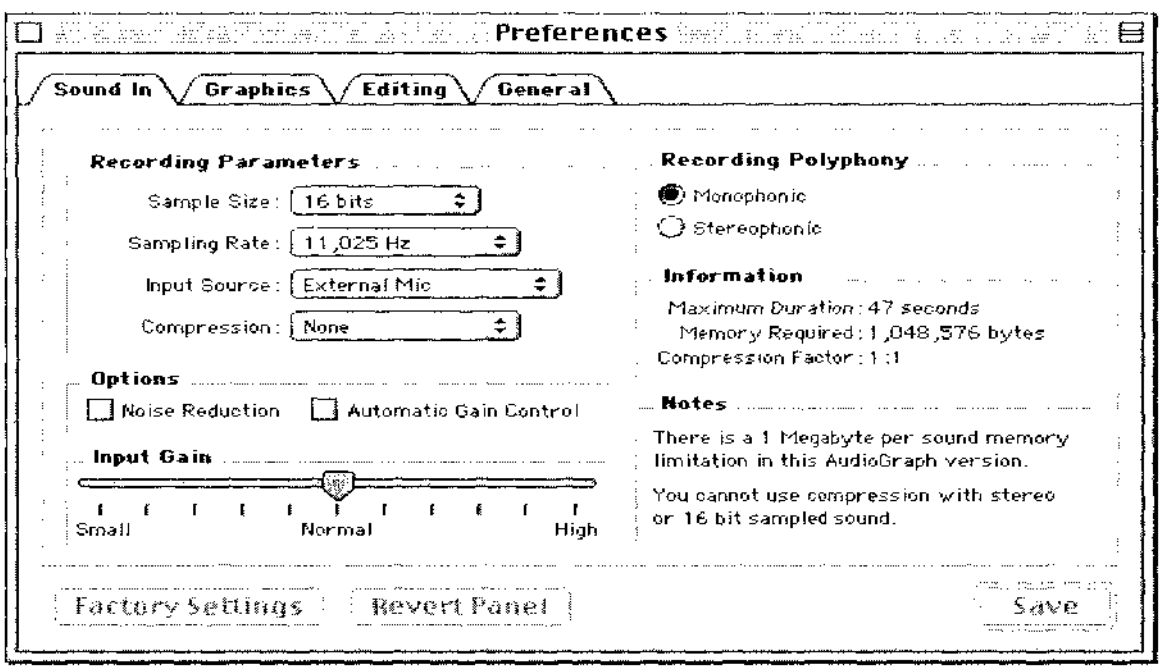


Figure 2-28. The Preference dialog

2.8 The evaluation of the interface of the AudioGraph Recorder

The AudioGraph has been used by a number of universities and companies for producing product training material, either on CDs or published on the Web, which has proved to be efficient and significant. The learning curve in using the AudioGraph Recorder authoring tool is shallow as the tool is based on the model of the lecture theatre or presentation room[C4].

Following the standard convention in the modern GUI, The AudioGraph Recorder interface presents users with a presentation language that is consistent, easy to follow and understandable. The techniques of the heuristic evaluation are used here for the evaluation for the AudioGraph Recorder interface.

Heuristic evaluation is the most popular of the usability inspection methods, and is done as a systematic inspection of a user interface design for usability. The goal of heuristic evaluation is to find the usability problems in the design so that they can be attended to as part of an iterative design process[B16]. This evaluation is performed by oral query amongst university teachers and students who have some computer knowledge.

In general, after studying the simple user guide, most users can fulfil basic tasks, such as importing a presentation and annotating with different drawing tools. Some problems have arisen however:

- Some icons were not familiar to a number of people, including such as the recording button and the picture button in the Tool Window, and some buttons in Edit Console.

  Solution: adding ToolTips for all buttons. As the mouse is over a button, a ToolTip will appear.

- Most users find the difficulty in using the Stealth button in the Attributes window, the Group/Regroup button in Edit Console, and how to get a Highlight Group and how to get a long duration sound annotation done.

  Solution: completing the file help system.

- Some users complain about the buttons in the Slide Window, because in Windows the control button should be put in a toolbar.

  Solution: as a compromise, it remains.

Some users with programming experience observe the performance of the application a little down, as for each Slide Window the Edit Console must create a set of annotation icons. Also each Slide Window has its own button states shown in the Tool Window and Attributes Window. When shifting from one Slide Window to another, all the states of buttons in the Tool Window and the Attribute Window must carry out corresponding changes. The Edit Console window must unload the annotation icons first and then load the icons in corresponding to the active Slide Window. All these are time consuming.

Part 3 Analyse the interface with UML

2.9 The analysis of the AudioGraph Recorder interface with UML

In the field of software engineering, object-oriented analysis and design is playing a leading role compared to the traditional development methods, such as structured analysis and structured design (SA/SD). The OO development process is used in all stages of the AudioGraph development. The notations used in the project are modelled by using the Unified Modelling Language (UML), and the adopted tool is Rational Rose.

2.9.1 UML

UML is "the industry-standard language for specifying, visualising, constructing, and documenting the artefacts of software systems. It simplifies the complex process of software design, making a 'blueprint' for construction"[B17].

UML is a unification of the concepts of previous object-oriented design notations. It combines the features of some popular previous system analysis and design methods, such as BOOCH, JACOBSON, RAMBAUGH, and it adds concepts and notations that have proved to be useful but were missing from those methods. It provides different views to perceive a system domain during development process. The views are presented by well-defined diagrams, which are used in the project and are classified as listed below:

- Functional view – this describes the system functional requirements. UML uses case diagrams and activity diagrams to depict a static functional view and a dynamic functional view separately.
- Static structure view – this defines the overall structure of the system. Class diagrams show the relationship of the classes and objects in the structure. An object diagram depicts a particular configuration in that it shows a set of objects and links at a specific moment in time during the execution of the system[C7].
- Behavioural (dynamic structure) view – UML interaction diagrams are used to show the transition behaviour of the system, such as sequence diagrams and collaboration diagrams. State transition diagrams present the status changes of an object. (As the use interaction and the Recorder interface have been described in the above section, and we have the current program as specification, this view is not given in the below section.).

2.9.2 Rational Rose

Rational Rose is Rational's market-leading Unified Modeling Language (UML)-based software modelling tool for designing component-based applications. Rose gives

developers the ability to mix and match multiple languages, such as C++, Visual Basic, and Java, within the same model[B18].

The application's method recommends the use of static and dynamic views of a logical model and a physical model to capture the in-process products of object-oriented analysis and design. Using the notation, the application enables users to create and refine these views within an overall model representing the user's problem domain and software system.

2.9.3 The analysis of AudioGraph Recorder interface with UML

Based on the scenario and the definition of the Macintosh AudioGraph Recorder, and the descriptions in the previous sections, the diagrams shown in this section identify the system functionality and the interface objects transition and behaviours.

Use case diagram

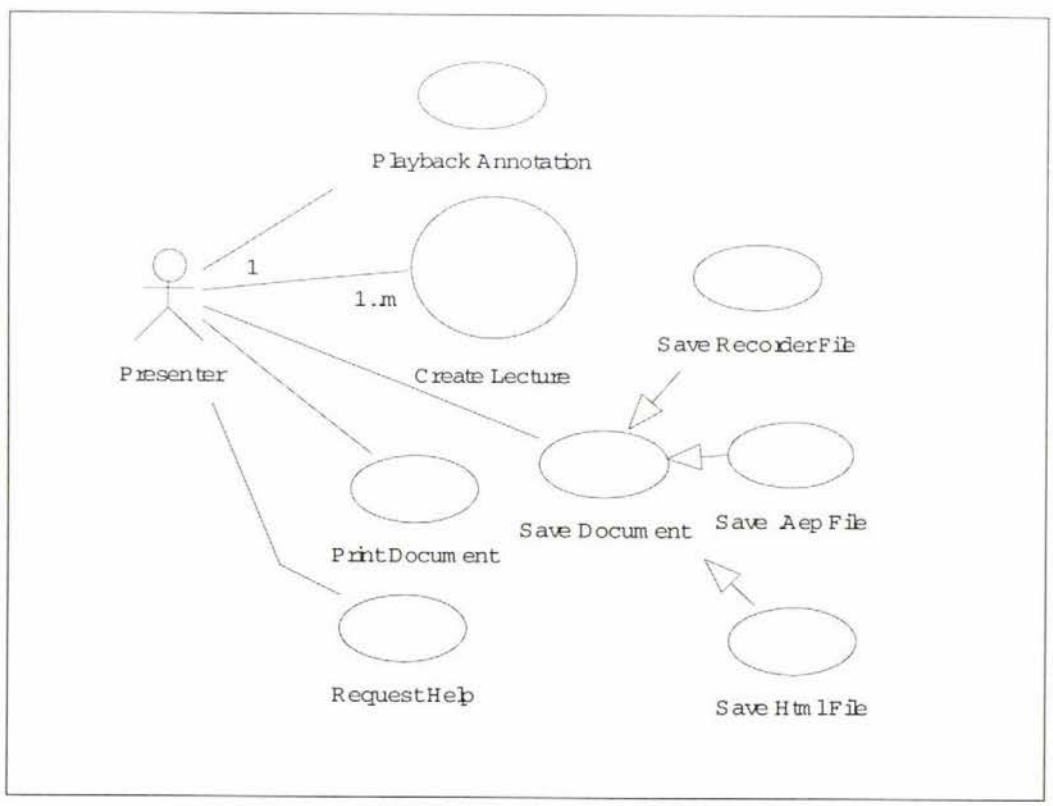


Figure 2-29. The use case diagram of AudioGraph Recorder

There is only one kind of actor (**Presenter**) who wants to create lectures using The Recorder. The use case diagram in Figure 2-29 shows all the functionality (use case) that the Recorder system can provide when a Presenter tries to use the AudioGraph Recorder, as well as the relationship between an actor and use cases.

Activity diagram

The main function provided by the Recorder is to create lectures. The use case diagram just shows a general use case (Create Lecture). To refine the Create Lecture use case, the following activity diagram is given to show the workflow of activities corresponding to the Create Lecture use case.

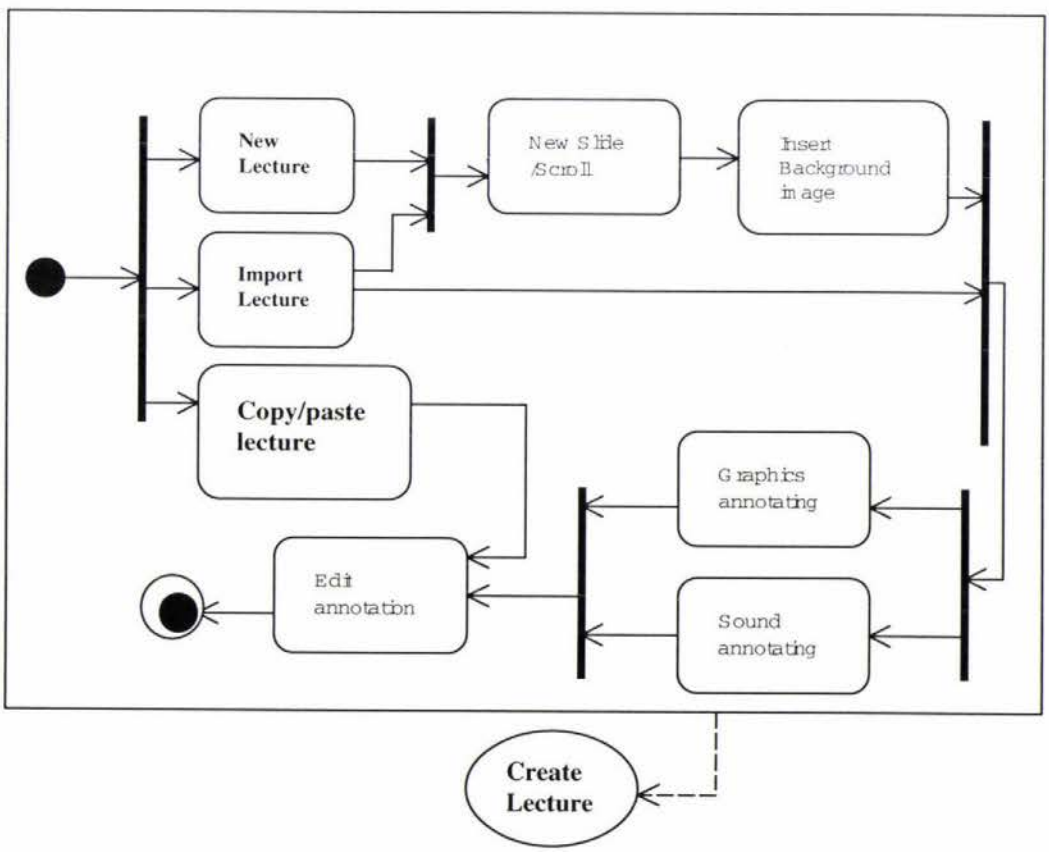


Figure 2-30. The activity diagram for creating a lecture

Class and Object Diagram

The class diagram is the most important diagram in OO development as it depicts the general structure of the system. The class diagram in Figure 2-30 for the Recorder interface shows the relationships among the interface main classes. The interface class information comes directly from the resource of the Macintosh AudioGraph Recorder, which is used in the PowerPlant framework in CodeWarrior environment. The class name uses the PowerPlant class convention.

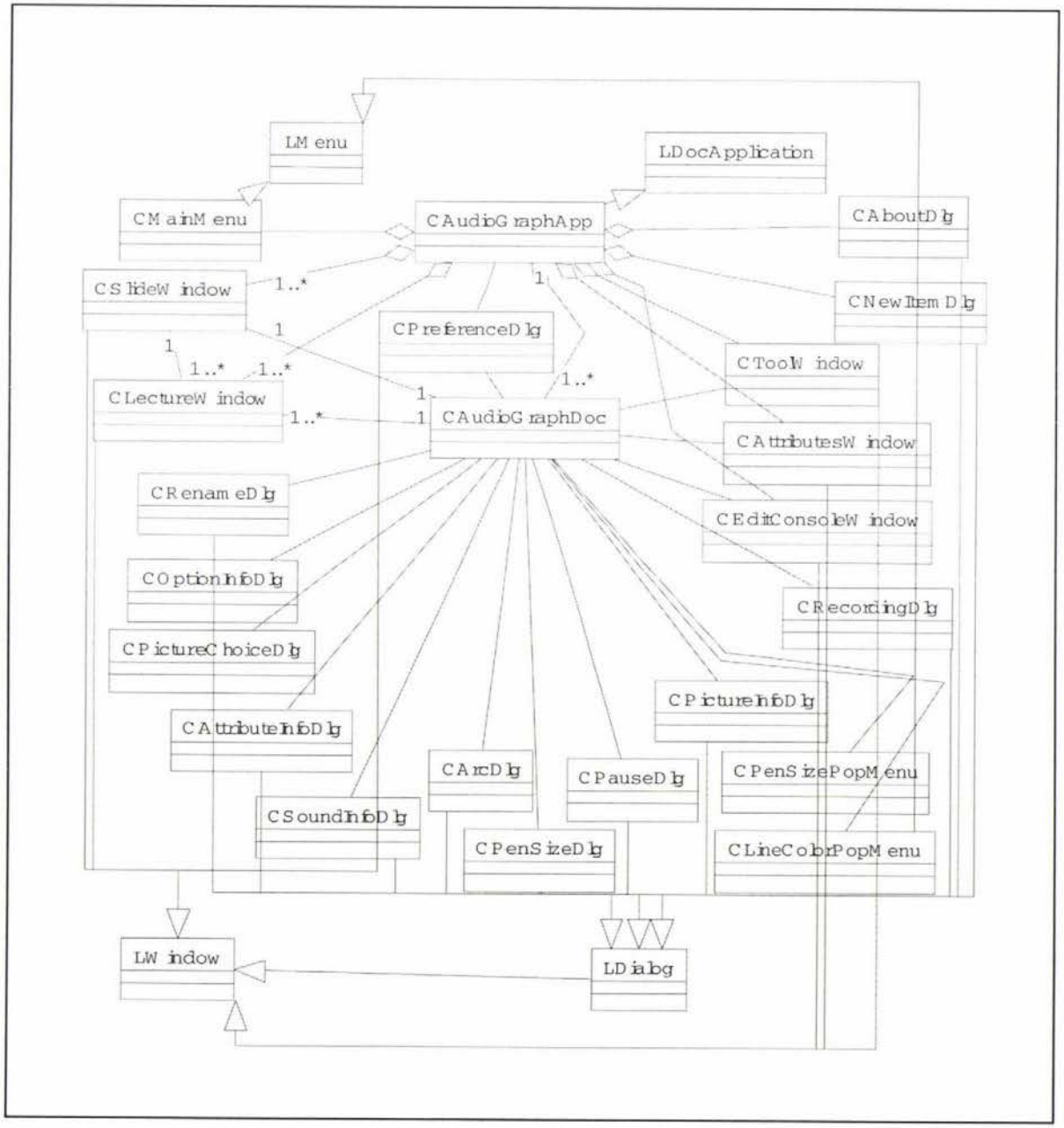


Figure 2-31. The class diagram of the AudioGraph Recorder Interface

The Recorder interface consists mainly of five windows: the Lecture Window, the Slide Window, the Tool Window, the Edit Console, and the Attributes Window. The following object diagrams show the objects contained in each of the four main interface windows. The object name uses the convention name of the PowerPlant classes. Note: if no multiplicity is indicated for the relationship between two instant objects, it means one to one relationship.

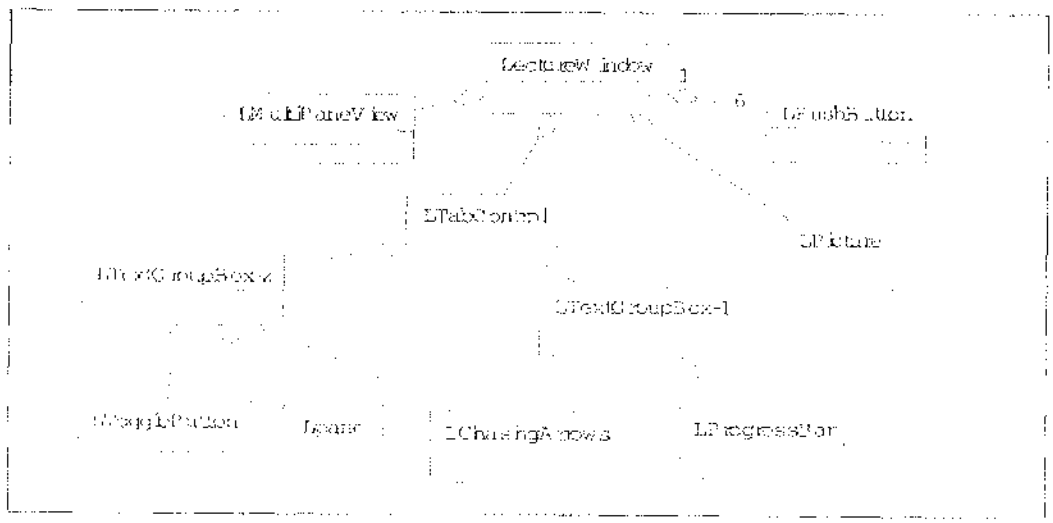


Figure 2-32. The object diagram of the Lecture Window

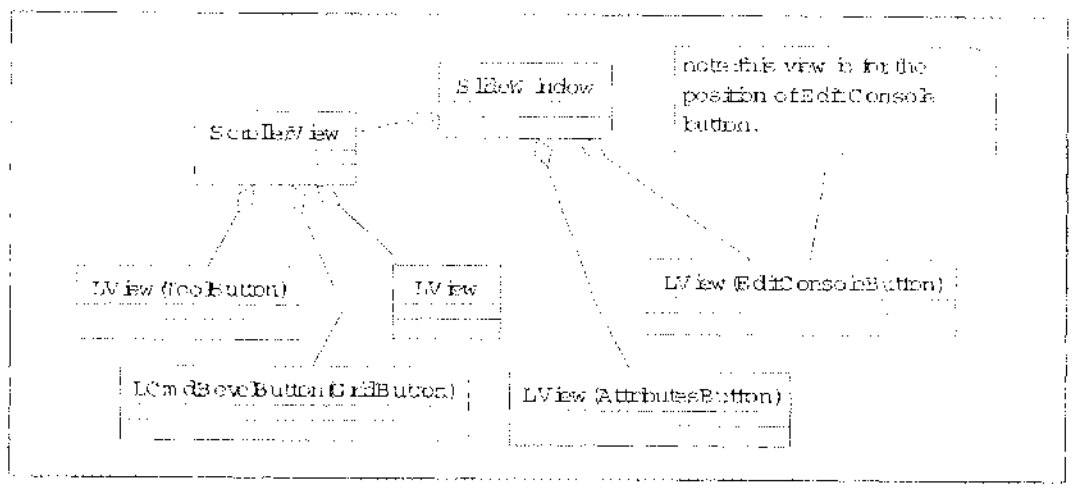


Figure 2-33. The object diagram of the Slide Window

Note: in the Slide Window, the AudioGraph Recorder uses a view object as a container of a button, such as a Tool button for control the Tool window.

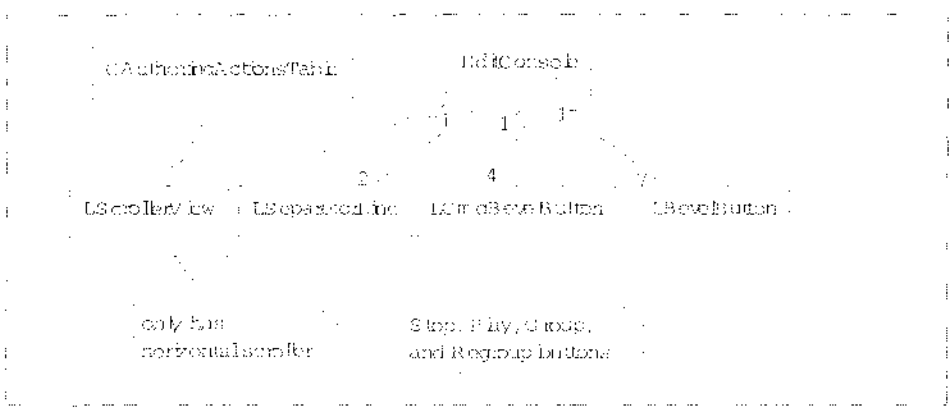


Figure 2-34. The object diagram of the Edit Console window

Note: All the buttons in the Edit Console have the same size. The CAuthoringActionTable is used for containing the inserted buttons when an annotation is done.

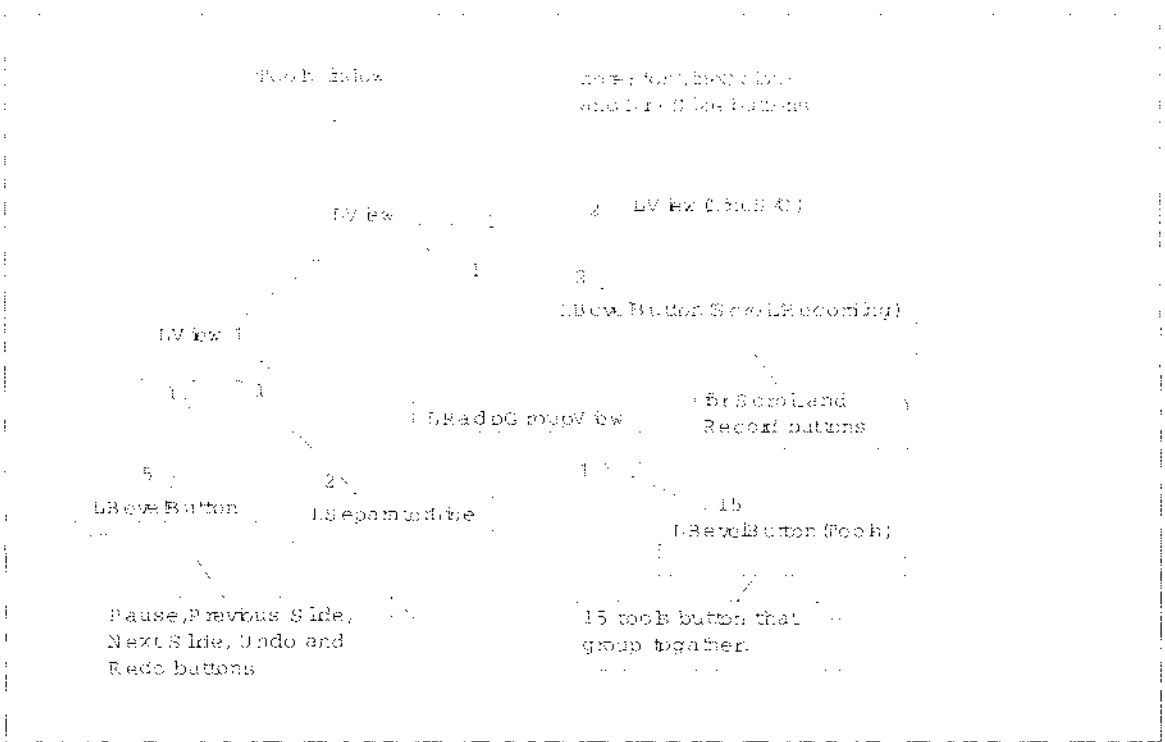


Figure 2-35. The object diagram of the Tool window

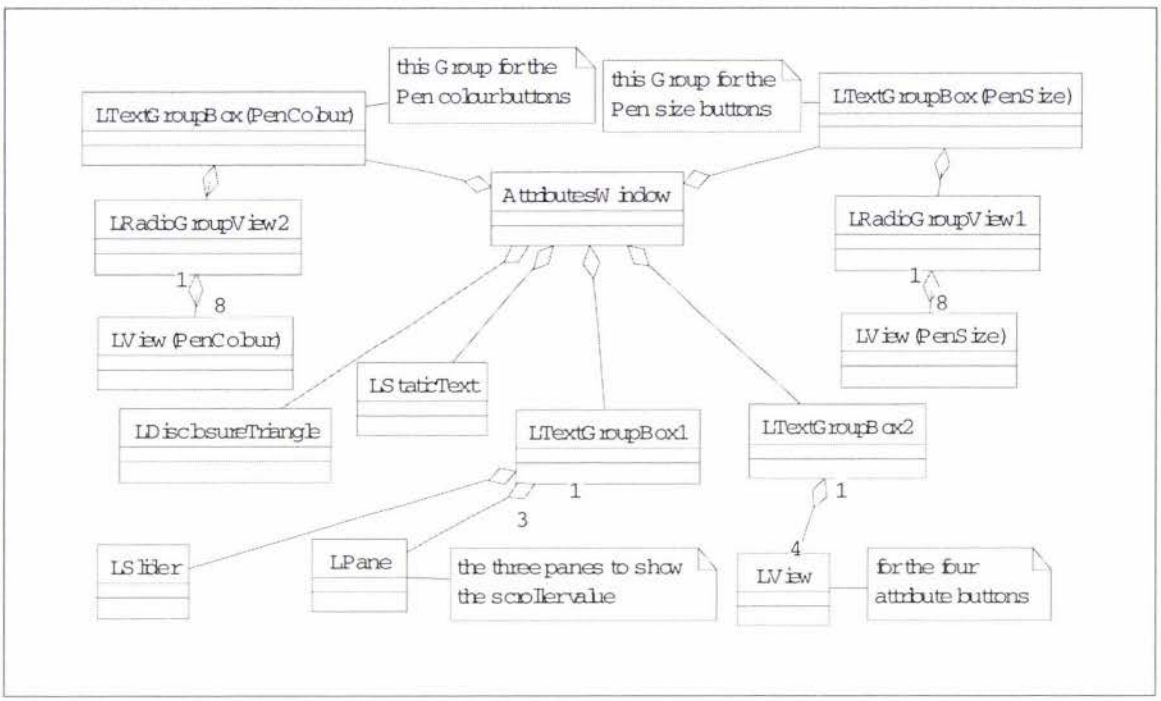


Figure 2-36. The object diagram of Attributes Window

The class diagram and the object diagrams will be used as references for the design of the PC version AudioGraph Recorder interface. As the Windows application has a main frame but the Macintosh doesn't, some modification will be made in the class diagram in Chapter 5. As the common Windows GUI components are similar to those in the Macintosh GUI, a component class in the PowerPlant framework can be mapped to a component class in MFC. So when we design the interface, we just map the class objects in the object diagrams into similar MFC class objects.

Summary

In this chapter, we have discussed the GUI differences between Windows and the Macintosh, and the Macintosh AudioGraph Recorder interface, and analysed the interface using UML. These analyses give the specifications of the application, which are used to guide the application design and implementation.

Chapter 3 Porting the AudioGraph Recorder from Macintosh platform to PC platform

This project is to build up a PC version AudioGraph Recorder based on the existing project – the Macintosh version of the AudioGraph Recorder. Although the project is particularly about the implementation of the interface part, the functionality of the system must be considered also to be compatible with the interface part also.

This chapter will discuss the difference between the development tools used in Mac OS and Windows—the PowerPlant and MFC application frameworks, followed by a discussion on the general issues about the porting of software applications. Based on the functionality and interface of the Macintosh version of the AudioGraph Recorder described in chapter 2, this chapter will define what functionality and interface components can be ported from the Mac platform to the PC Platform or need to be redeveloped for the PC platform.

Part 1 Comparing PowerPlant with MFC

PowerPlant and MFC are both C++ application frameworks. They both provide a collection of classes to support visual components in GUI and event-driven programming that requires the handling of events. Also, as the AudioGraph Recorder is a document-based application, it needs to deal with documents. As application frameworks, PowerPlant and MFC have something in common with regards to the design patterns, but the implementation of them differs. While they both use fundamental Object-Oriented technologies such as inheritance to build their class hierarchy and add functionality to the frameworks, the design principles for them are different. PowerPlant uses multiple inheritance, whereas MFC uses single inheritance.

3.1 Design principle for inheritance

PowerPlant takes full advantage of C++'s support for multiple inheritance to provide a mix-in architecture and has a set of base or mix-in classes that are used to augment other classes in its class hierarchy. For example, any of the following classes may be inherited from:

- LPane—for visible objects.
- LCommander—for command objects.
- Lbroadcaster—for all objects that broadcast messages.
- LListener—for all objects listen for messages.
- LPeriodical—for objects that should receive time on a regular basis.
- LAttachable—for objects to which attachments can be connected.

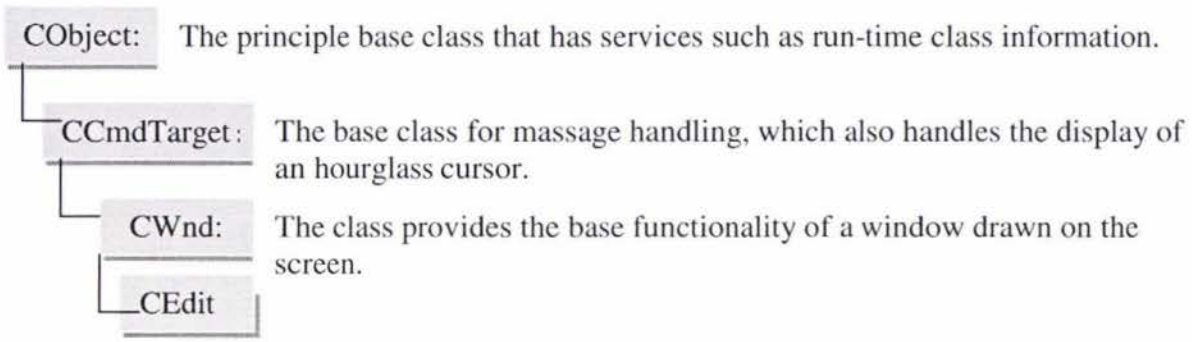
The class hierarchy in MFC is a monolithic tree where almost every class is derived from CObject class. For each layer in the single inheritance hierarchy, a certain functionality and behavior is added into the class in its top layer to form new classes.

The following is an example of an edit box to demonstrate how PowerPlant and MFC use inheritance differently.

PowerPlant has a LEditField class for an edit box. LEditField inherits directly or indirectly from:

- LPane—to draw on the screen.
- LCommander—to respond to messages such as keyboard input.
- LPeriodical—to get time for flashing the cursor when the field is focused.
- LAttachable—to change things such as the behaviors of other attached objects when an editable text field is present.

MFC however, has a CEdit class for an edit box that is part of the following class hierarchy:



In single inheritance design, as mentioned in Chapter 1, subclasses end up inheriting a lot of 'baggage' that they will never use. However, multiple inheritance adds the complex to the implementation of an application. Using multiple inheritance design, a class is far more likely to have just the behavior it needs.

In general, PowerPlant and MFC both provide similar a class hierarchy to support GUI and to provide their own mechanisms to deal with event handling for window programming. They also support document-based applications to deal with data saving, and retrieving to and from files. Some differences do exist in these domains however and they are described next.

3.2 Framework implementation

Although the details of solutions vary from framework to framework, most frameworks do follow some common design patterns as all applications deal with similar problem domains. Frameworks have a group of classes to accomplish some specific tasks.

As a framework to deal with event-driven programming, PowerPlant and MFC have followed similar design patterns [A7], which are listed below;

- *Applications*
- *Event handling*

- *Visual hierarchy*
- *Persistence*

PowerPant and MFC have their own classes and architectures to make them application frameworks. Following the common design patterns, the following discussions will expose some differences between PowerPant and MFC regarding how they solve these high-level problems.

3.2.1 Applications

PowerPlant and MFC frameworks both have an application object. This object provides application-level services to hold an application together. It is responsible for launching and quitting the application, initializing the environment, and running a main event loop. It also handles document creation and events that are not handled by other objects such as windows.

The main event loop, which is hidden underneath PowerPlant or MFC, is essential to event driven programs. In both frameworks it repeatedly retrieves events until a quit-message is encountered to quit the application. After retrieving an event, the main loop then parses, translates the event before finally dispatching it to a target object.

PowerPlant has two classes for the application objects, LApplication and its derived class (LDocApplication). LDocApplication has additional functionality than LApplication to support opening, closing, and printing documents. MFC has only one application class – CWinApp. The differences between the application objects of PowerPlant and MFC, is that CWinApp is responsible for creating at least one window (remembering that a Windows application must have a main frame, whereas a Mac application doesn't), and CWinApp is not responsible for creating the main menu. This is taken care of by the main frame object of an application.

3.2.2 Event Handling

Both Windows and Macintosh applications are event driven, so PowerPlant and MFC must handle events, which they do differently from each other. Whereas PowerPlant uses a command hierarchy and a broadcaster-listener system to handle events, MFC uses a message-mapping mechanism which was briefly mentioned in Chapter 1.

PowerPlant

PowerPlant has a LCommander class that is the base class from which all commander objects inherit. A commander is an object that can handle events and respond to commands. A command is a menu selection or command-key equivalent. The LCommand class has functions for command chain maintenance. In the command chain, those higher up are called super-commanders and those lower down are called sub-commanders. The command hierarchy is a tree, in which no object has more than one super-commander, but it may have zero or several sub-commanders.

PowerPlant dispatches events from the bottom up using the command hierarchy. In the design pattern of PowerPlant, the application keeps track of a target object, which is the currently active command object destined to be the recipient of all events. An event is directly dispatched to the target object. However, if the target is incapable of handling the event, it passes the event back up the command hierarchy to its owner/super-commander, as shown in Figure 3-1:

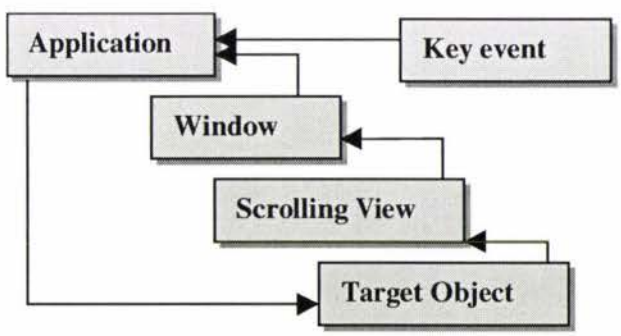


Figure 3-1. Bottom-up command hierarchy

Fundamental to any commander's behavior is its ability to handle commands and keystrokes. The LCommander class has three principle functions that can be overridden to handle commands and keystrokes:

- ObeyCommand() when a menu item is selected, the target commander's ObeyCommand() member function receives an identifying command number referring to an particular menu item, and is where programmer writes codes to respond to the event.
- FindCommandStatus() this function is to manage the states of the menu items, such as to enable , disable, and mark menu items.
- HandleKeyPress()—this function receives the message as a key is pressed and responds accordingly.

PowerPlant also has another messaging system, the broadcaster-listener system. This enables objects to communicate with each other when they are not part of the same chain of commanders, the same command hierarchy, or when the message involved is not an event.

An object inherited from LBroadcaster in PowerPlant, has the ability to broadcast a message. All the control classes derived from LBroadcaster, such as a check box, may send a message when it is clicked. An Object inherited from LListener has the ability to receive a message from a broadcaster. When a broadcaster wants to send a message to some objects, it uses the member function AddListener() that links new listeners to itself. When a listener gets a message, its member function ListenToMessage() is called as the event handler, to handle the event.

MFC

In MFC's message-handling architecture, there are two components which work together to handle messages. They are: 1) the CCmdTarget class and 2) message maps. For a class to handle messages it must be the descendent of the CCmdTarget class. In MFC class hierarchy all the application classes, document classes, and all visual components such as window classes, are derived from CCmdTarget class.

In MFC, there are three main categories of messages that the corresponding handlers need to deal with:

1. Windows messages

   This includes primarily those messages beginning with the WM_ prefix, except for WM_COMMAND. Windows messages are handled by windows and views. These messages are standard Windows messages, such as WM_LBUTTONUP which signals that the left mouse button has been released.

2. Control notifications messages

   This includes WM_COMMAND notification messages from controls and other child windows to their parent windows. For example, when the user has altered text in an edit control, the edit control sends its parent a WM_COMMAND message containing the EN_CHANGE control-notification code. The window's handler for the notification message may need to deal with retrieving the text in the control.

3. Command messages

   This includes WM_COMMAND notification messages from user-interface objects: menus, toolbar buttons, and accelerator keys. MFC defines unique identifiers for standard menu and toolbar command message.

Messages in categories 1 and 2, that are Windows messages and control notifications, are handled by windows: objects of classes derived from class CWnd. This includes CFrameWnd, CMDIFrameWnd, CMDIChildWnd, CView, CDialog, and the classes derived from these base classes. Command messages in category 3 can be handled by a wider variety of objects: documents, document templates, and the application object itself in addition to windows and views.

For all command messages, MFC also has a built-in routing architecture to handle the case if one class can't process a command message. For the MDI program case of the AudioGraph Recorder application, the sequence in which classes are offered an opportunity to handle a command message is as follows:

1. The active view object.
2. The document object associated to the active view.
3. The document template object related to the active document, which is part of the document-view architecture.
4. The frame window object that contains the active view.
5. The main frame window object.
6. The application object.

The active view is given the chance to handle a command message first. If it has no handler to deal with the message, the next class object has the chance to process it. However, if none of the objects can handle the command message, then the message will be passed to the default Windows processing to throw the message away.

The comparison of event handling between PowerPlant and MFC

PowerPlant and MFC both have their bottom-up command hierarchy to simplify the command dispatching problem. The command hierarchy in MFC is fixed, which means that the class objects in the command hierarchy have a pre-defined priority to handle commands. In PowerPlant however, LCommand has functions to allow an application to maintain its command hierarchy and set the target object to first receive a command. In this way, the command hierarchy in PowerPlant is more flexible.

In PowerPlant, message handlers are pre-defined member functions of a class, such as the ObeyCommand() member function in the LCommand class, that always has a *switch* statement to handle different messages. In MFC, every handler is a member function of a class that just responds to one message. The message handler in PowerPlant may grow very large if it has too many messages to deal with.

The broadcaster-listener messaging system in PowerPlant is much more flexible than the message mapping in MFC, because a broadcaster can have multiple listeners and doesn't need to know anything about its listeners.

For example, if a check box is clicked, a radio button will be disabled. In PowerPlant, programmers just need to add the radio button object as a listener of the check box object using AddListener() function. Some code can then be added to disable the radio button object in its member function ListenToMessage(). However, in MFC, the handler for the clicking message of the check box needs to get the radio box object first, then call the radio button's proper member function to disable the radio button.

The disadvantage for the broadcaster-listener messaging system is the need to implement a class for each listener, such as a class for the radio button in the above example.

From a high level view of CPU processing, the broadcaster-listener messaging system breaks the processing responding to an event into several processes if the event has several listeners respond. During the CPU processing of the event, some input occurs which needs to be dealt with first by the CPU, in this case, the CPU can quickly drop the current process.

3.2.3 Visual hierarchy

PowerPlant and MFC both have a class hierarchy to support visual interface components such as windows and dialogs etc. The following explains some major classes used in visual class hierarchy for both PowerPlant and MFC.

PowerPlant

In PowerPlant, LPane is the base class for all the display components. A pane represents a rectangular drawing area.

➢ *LPane*

LPane is the fundamental display object which describes a rectangular area that can display graphics and responds to mouse clicks. Every pane belongs to a superview and does not have their own coordinate system; they use the coordinate system of their superview. Panes cannot have subpanes, and are not scrollable.

The derivatives of LPane class are:

- LCaption—display static text.
- LGroup—display bounds of a group items. This class derives from the LCaption.
- LIconPane—display an icon.
- LMovieController—display a QuickTime movie.
- LEditField—display an editable text box.
- LListBox—display a Macintosh List Manager list. This class can be used to create a two dimensional table of cells that scroll, respond to keystrokes, and so forth.
- LBevelButton–a rectangular control with a beveled edge that gives the button a three-dimensional appearance. It can display text, an icon, or a picture. LcmdBevelButton is derived from LBevelButton and LCommand, which can be part of the command hierarchy.
- LDisclosureTriangle–used to provide a way for users to expand a dialog box or control panel. When the user clicks on the disclosure triangle, the triangle rotates downward and the window expands to provide supplemental information.
- LPopupButton–k displays a list of items the user can select to change the state of an aspect of the application. A pop-up menu button consists of a single control. The left side of the button contains text that shows the current selection; the right side of the button shows a double triangle pointing up and down.
- LPushButton—it is a rounded rectangle that is labeled with text.
- LSeparator—used in dialog boxes to separate groups of controls by delineating horizontal or vertical regions of the content area.

- LStaticTex t—embeds static text (unchangeable by the user) in dialog boxes.
- LTextGroupBox—used to associate, isolate, and distinguish groups of related items in a dialog box.
- LChasingArrow—is a pane which is used as an animation control.

There are two important classes derived from LPane: LView and LControl. As these two classes have lots of derivatives, separate introductions are given below.

➢ ***LView***

LView is derived from LPane. Views have their own coordinate system and can hold subpanes. In addition to the frame that LView inherits from LPane, views define a rectangular area called the image that allows views to scroll their contents. The image is a potentially large one, only a portion of which appears in the frame. Scrolling entails moving the image so that different parts appear in the frame.

The derivatives of LView class are:

- LScroller—A scroller is a view that contains scroll bars and a view to be scrolled. LScroller can manage the communication between the scroll bars and the view.
- LTable—displays and manages tabular data.
- LPicture—displays a PICT resource.
- LPrintout—is associated with a printer port. To print panes that appear in a window or a GWorld, PowerPlant moves the panes to a LPrintout object.
- LPlaceholder—holds a rectangle for the size and position of other objects.
- LTextEdit—displays and manages editable text.
- LWindow—represents a window on the screen. A window can be made into a floating window or a modal window.
- LDialogBox—is derived from LWindow. A dialog box is a specialized kind of window that may have a default button and a cancel button.

- LMultiPaneView— is a view that can associate with different views in a tab control.
- LTabsControl--provides a convenient way to present information in a multi-page format and it works together with LMultiPaneView.
- CThumbnailsTable—it is a view used to display thumbnail icons.

➢ ***LControl***

LControl is derived from LPane and is an abstract class for objects that store a value and do something when clicked. LControl is derived from LPane and LBroadcaster. When a control's value changes, it sends a message to its listeners.

The derivatives of LControl class are:

- LButton—has a graphical element as the visual representation of a button.
- LCicnbutton—is identical to LButton but its graphical element uses the Mac OS 'cicn' resource format.
- LToggleButton— is identical to LButton but its graphical element are ICON, ICN#, or PICT graphics.
- LTextButton—is a button using textual representation.
- LStdControl—is a wrapper class for the Macintosh Control Manager. It encapsulates the standard behavior of a Mac OS control item. Three derived classes, LStdButton for the standard push button, LStdCheckBox for check box, LStdRadioButton for radio box, implement the standard behavior of the built-in controls.
- LProgressBar—is used to inform the user about duration or capacity.
- LSlider—is used to display a range of allowable values. It has an indicator to mark the current setting.
- LScrollBar—is used for users to view areas of a document or a list that is larger than can fit into the current window.

MFC

In MFC, CWnd is the class from which all other window are derived. It is derived from CCmdTarget, which enables it to receive and handle messages. CWnd wraps most of the common API functions that operate on a window. The following diagram shows the six main categories derived from CWnd:

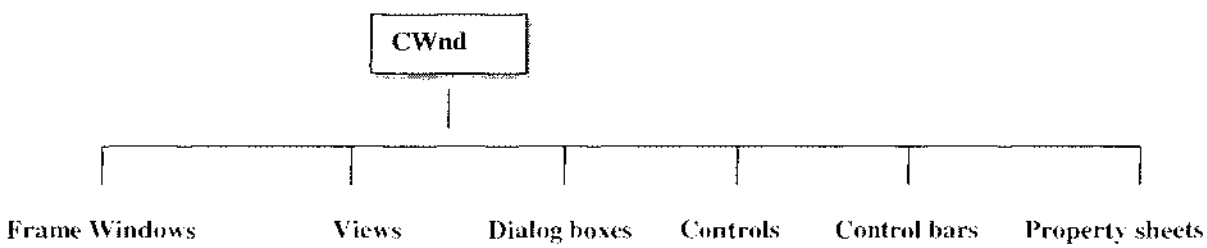


Figure 3-2. CWnd class and its derivatives

The following will explain some of the classes in the six categories.

1. Frame windows

Each application has a main window, and each document application has a document frame window. The frame window class objects are the frame of windows, such as main window frame. The classes for frame window are as follows:

- CFrameWnd -- is used as the main windows for SDI applications.
- CMDIFrameWnd -- is used as the main window for MDI applications.
- CMDIChildWnd -- provides child windows for MDI applications.
- CMiniFrameWnd -- A CMiniFrameWnd object represents a half-height frame window typically seen around floating toolbars. These mini-frame windows behave like normal frame windows except they do not have minimize/maximize buttons.

2. *Views*

A view is a rectangle that appears on the screen, which is used as a child of a frame window. MFC uses the document/view architecture to separate actual data from representations of that data. A view is attached to a document and acts as an intermediary between the document and the user. The view renders an image of the document onto the screen or printer and interprets user input as operations upon the document. Some views in MFC are:

- CVew—The CView class provides the basic functionality for user-defined view classes. The view is responsible for displaying and modifying the document's data but not for storing it.
- CScrollView—The CScrollView class derived from CView with scrolling ability. It has a vertical scrollbar and a horizontal scrollbar.
- CFromView—it derived from the CSrollView melded with a dialog template. It can have many controls as dialog box.

3. *Dialog boxes*

CDialog class is used as the base class to support dialog boxes. Derived from the CDialog class, MFC provides some common dialogs such as, CFileDialog for selecting a file from a directory, and CColorDialog for selecting a specified color. There is a class CPropertyPage derived from CDialog. Objects of class CPropertyPage represent individual pages of a property sheet, otherwise known as a tab dialog box.

4. *Controls*

The following lists some common control classes in MFC:

- CButton and CBitmapButton—a standard Windows pushbutton and custom pushbutton with a bitmap.

- CComboxBox-- A combo box consists of a list box combined with either a static control or edit control. The list-box portion of the control may be displayed at all times or may only drop down when the user selects the drop-down arrow next to the control.
- CEdit—An edit control is a rectangular child window in which the user can enter text.
- CScrollBar—standard Windows scroll bar.
- CListBox— The CListBox class provides the functionality of a Windows list box. A list box displays a list of items.
- CStatic— A static control displays a text string, box, rectangle, icon, cursor, bitmap, or enhanced metafile.
- CAnimateCtrl—a control that plays animations.
- CProgressCtrl—Display a progress.
- CSlideCtrl—a slider for choosing between a range of value.
- CStatusBatCtrl-- A "status bar control" is a horizontal window, usually displayed at the bottom of a parent window, in which an application can display various kinds of status information.
- CToolBarCtrl-- A Windows toolbar common control is a rectangular child window that contains one or more buttons. These buttons can display a bitmap image, a string, or both.
- CListCtrl—display a graphical list of list items.
- CTabCtrl—for Windows common tab controls.

5. Control Bars

A control bar is a window that is usually aligned to the left or right of a frame window. Control-bar windows are usually child windows of a parent frame window and are usually siblings to the client view or MDI client of the frame window. CControlBar is the base class for the control-bar classes:

- CStatusBar— a status bar, items are status bar panes containing text.

- CToolBar—a toolbar, items are bitmap buttons aligned in a row.
- CDialogBar—a toolbar-like frame containing standard windows controls (created from a dialog template resource).

6. Property Sheets

Objects of class CPropertySheet represent property sheets, otherwise known as tab dialog boxes. A property sheet consists of a CPropertySheet object and one or more CPropertyPage objects. It is displayed by the framework as a window with a set of tab indices with which the user selects the current page, and an area for the currently selected page. Even though CPropertySheet is not derived from CDialog, managing a CPropertySheet object is similar to managing a CDialog object.

Compare PowerPlant with MFC in visual hierarchy

PowerPlant and MFC provide similar classes in visual hierarchy. Some modern standard interface components found in PowerPlant can be mapping into the classes in MFC, such as push button, list box, tab control, dialog etc.

MFC provides custom visual components classes, such as CFontDialog, and provides control bar classes to support control bars like toolbar, but PowerPlant does not have such classes.

The classes in PowerPlant are relatively more simple than MFC, as PowerPlant uses multiple inheritance. The visual hierarchy in PowerPlant was designed to start with the basic interface component, a rectangle area, then gradually builds up other components by adding functionality. MFC in contrast, first starts with CWnd to wrap most Windows API functions for the operations on windows, then adds functionality to provide other classes in the class hierarchy. From the point view of Object-Oriented design, the visual hierarchy for PowerPlant is a more sensible one than that of MFC.

3.2.4 Persistence

Persistence refers to the concept that an objects content should continue to exist even when the application is not running. It also refers to the ability to store objects and recover their state later from a medium such as hard disk.

At a high level, PowerPlant and MFC have common solutions to persistence. They both use file classes to manage file operations such as, opening and closing a file, a document to hold object data, and a binary stream to manage the data moving around. A stream is an ordered series of bytes of data.

In PowerPlant, there are the following classes to support persistence:

- LStream— is the base class for a stream operation, such as reading and writing data.
- LFile— provides functions for opening, closing, reading, and writing both the data and resource forks of a file.
- LFileStream—inherited from LSteam and LFile. It has operations to stream data into or out of a file.
- LDocument— is responsible for writing and reading its content to and from storage by interaction through the LFile object.

Dealing with object persistence with these classes, programmers still need to do a lot of work. An example of what a programmer basically needs to do for a line object that needs to be saved and restored from a file in an application with PowerPlant is as follows:.

To save a line object in a file, in the DoSave() of the document classes the following functionality must added:

- Open a file.

- Store the line object type and the line object attributes such as start_Point and End_Point in a file.
- Close the file.

To open a file (suppose it only contains a line object), a programmer needs coding to:

- Open the file.
- Read all the content from the file.
- Distinguish that it is a line object by the line object type.
- Create the line object depending on its attributes.
- Close the file.

In MFC, a programmer doesn't need to worry about all these as it has a serialization mechanism to take care of the object saving and restoring. Being derived from CObject, a class has the ability for serialization. To support serialization in a class derived from CObject, a programmer needs to:

- Apply the macro DECLARE_SERIAL and IMPLEMENT_SERIAL in the class definition and implementation parts.
- Override the CObject::Serialize() method to write/read the classes member data. For example:

```
Void CLine::serialize( Carchive & ar)
{
    if (ar.IsStoring()){
            ar<<Start_Point; //Start_Point is one member of CLine
            ar<<End_Point; //End_Point is one member of CLine
    }
    else{
            ar>>Start_Point;
            ar>>End_Point;
    }
}
```

To save and restore an object contained in a document, the document class just needs to override the inherited CDocumetn::Serialize() method to insert and extract the object.

In order to support persistence in MFC, it also has a CDocument class. However, one difference with PowerPlant, is that MFC has a help class, CArchive class, which implements a binary stream. The CArchive class is tied to a pointer of a file class CFile and implements the buffering of data. It encapsulates the functionality of file operation, object writing and reading.

From the point of view of a programmer, MFC provides a much simpler method of dealing with persistence than PowerPlant does.

Part 2 Porting software applications in general

Portability is becoming universally recognized as a desirable property for most software products, and porting is a recognized technique for extending the value and life of a software unit.

Porting is the act of producing an executable version of a software unit or system in a new environment, based on an existing version [C8]. The term *software unit* is used to indicate an application program, a system program, or a component of a program. A software system is made up of a collection of software units. The term *environment* (or *platform)* refers to the complete range of elements that the ported software interacts with, and mainly refers to operating systems (such as Window NT, Mac OS 8.5). It may also involve a processor, I/O devices, libraries, or networks.

The term *portability* refers to the ability of a software unit to be ported (to a given environment). The main factor that affects the portability of a software unit is the cost. Depending on the cost, sometimes redevelopment is required. In most cases the porting process and the redeveloped are overlapped.

The portability of software and the porting techniques are playing very important roles in software engineering as there is a growing need for extending the use of current existing software. This is due particularly to the reasons:

- Hardware architectures being updated so fast.
- Software concepts and programming language keep progressing to suit user's easy-use demands.
- In the commercialized society the "universal computing environment" is unlikely to become a reality, even though a few standard platforms (*e.g.* IBM-PC, Macintosh, Unix) have become widely used in some domains.

Due to these considerations, most software packages will have to finally face the need to be ported in order to maintain and expand their viability. This need can be most effectively met if portability has been "built in" during the development process. Therefore, the systematical evaluation of the portability of existing software in specific situations is important, and leads directly to the selection of an effective porting strategy to reduce the cost.

The principal types of portability usually considered are *binary portability* (porting the executable form) and *source portability* (porting the source language representation). Binary portability is possible only across strongly similar environments, which can't be satisfied in most situations. Source portability happens all the time, which provides opportunities to adapt a software unit to a wide range of environments. In this project, the strategy of source portability is used.

The porting process has two principal components: *transportation* and *adaptation*. Transportation is physical movement, which may require some conversion tools. Adaptation is any modification that must be performed on the original version, and may be separated into manual modification and automated translation. The two components are always used together.

3.3 The environment introduction of the AudioGraph Recorder

The Macintosh version Recorder has been developed by the application framework tool of PowerPlant in CodeWarrior 11: Integrated Development Environment (IDE). The operating system is Mac OS 8.5. The programming language used is standard C++.

The target environment is PC platform and the operating systems on PC refer to Windows 95, Windows 98, Windows NT, and Windows 2000. In all the modern Windows OS except MS DOS, Win32 is the main subsystem, which runs Win32 applications, and manages all keyboard, mouse, and screen I/O. Therefore, in this sense, the target environment is Win32. During the development, Windows NT was used as it was available in the Massey lab and it supports Unicode, which increases the portability of the application for multiple languages. (*Unicode* is an approach to handling large character sets, such as Chinese and Japanese. Normally 8-bit character encoding is used. Under *Unicode*, a character uses 16 bits for encoding).

The MFC application framework in the Visual C++ 6.0 programming environment was used as the tool for the development of the PC version. The programming language for Visual C++ is C++.

The currently available documentation for the AudioGraph Recorder is the general introduction—a user manual of the AudioGraph Recorder. The source code is also available. All the design and implementation documents must be extracted from the application source code.

3.4 Evaluate the portability of the Mac AudioGraph Recorder

As this project development on Windows uses the same programming language C++ as the one used on Macintosh, the code in the AudioGraph Recorder that can be ported to Windows are those platform-independent, in other words those which are not involved with using the Macintosh APIs.

According to the coding functionality of the AudioGraph Recorder on Macintosh, the software units can be mainly categorized as:

1. Creating interface and managing interface states;
2. Drawing and editing different shapes with different pen color and size;
3. Dealing with picture;
4. Dealing with sound;
5. Dealing with saving and retrieving data or objects;
6. Setting up html file format;

Although the PowerPlant framework is different to MFC, the APIs for dealing with interface components on Macintosh perform a similar functionality as the APIs in Windows. At a conceptual level, an API function of Macintosh can be "mapped" to Windows API function. With this point in mind, according to the functionality of the code, general evaluation of portability of the Mac Recorder project is discussed below.

3.4.1 Creating interface and managing interface states

The software units involved with interface, are about interface class's declaration and implementation parts. Each interface component in the Mac Recorder is implemented by a derived class from the PowerPlant framework class hierarchy. After setting up an instance of an interface component class, an interface can be created, appear on the screen and destroyed. The states of the object are manipulated through its member functions that heavily use the Macintosh APIs.

The class libraries of PowerPlant are incompatible with Windows as Macintosh API functions are used in these classes. For the development of the PC version application, the interface part must be redeveloped. But because the PowerPlant frame provides a collection of classes to support the common GUI, as MFC does, "mapping" from a class on the Macintosh to the similar one in Windows can be achieved. Although the implementation for PowerPlant and MFC is different, in both platforms the relationships

among the interface classes of the AudioGraph Recorder are the same, as are the events for each interface class. Event handling for each interface class is similar in functionality.

3.4.2 Drawing and editing different shapes with different pen color and size

The code units dealing with drawing are involved with drawing component classes such as line, rectangle, oval and curve, which have a common super-class. These components have some common behaviors, such as moving, drawing and hit-by etc, and some common attributes such as brush, color and pen size etc.

This code can be ported to Windows, although modifications are needed. Some data types such as point type and rectangle type must be modified, as well as the implementation of drawing methods for Windows. In Mac, drawing uses QuickDraw, which depends on the *graphics port* data structure to define pen size, color, font and draw area boundaries etc, whereas Windows uses *device context* for similar purposes. In order to support the MFC serialization, some functionality must be added in the component classes.

3.4.3 Dealing with picture

This code uses the QuickTime programming interface (API) to import, covert and compress still images. In QuickTime, "Graphic importer and exporter components are used to open, display and export graphic images stored using various file formats and compression algorithms"[B19].

QuickTime is a system-level code package that supports a variety of multimedia tasks. The QuickTime API is a large set of system-level C functions and associated data structures. "Although Mac and Windows programs may be structured differently, their interface to QuickTime are virtually identical. Just about anything QuickTime can do on one platform, it can do on the other." [A8]. However, the implementation for Windows is

through a dynamic-link library (DLL), while Mac uses QuickTime as a set of system extensions.

3.4.4 Dealing with sound

This code deals with the sound recording and sound playback. The Macintosh AudioGraph Recorder records sound through the Sound Input Manager, "which provides a number of routines for the assistance to begin, pause, resume, and stop recording directly from a sound input device" [B20]. To playback sound is through the Sound Manager. "The Sound manager uses an architecture based on sound components to process audio samples for playback. A sound component is a software module that performs a specific task, usually some kind of audio processing like decompression, rate conversion, sample format conversion, or mixing" [B21].

This code can't be ported to Windows because Windows has its API functions for waveform audio format to deal with sound recording and playback[A10]. Waveform audio is sampled, digitized audio data, which can be stored in file with the .wav extension.

3.4.5 Dealing with saving and retrieving data

Data in the AudioGraph Recorder are objects such as rectangle, point, image, and circle etc. PowerPlant and MFC both use a document to manage data for an application. Although PowerPlant and MFC have different ways of dealing with the saving and retrieving of data, they deal with the document class in a similar way to each other. The document must have some data structures as data members to keep all the data or objects.

This code for the data structure in a document, therefore, is portable in which some data type may need to be modified, such as string, point and rectangle type etc.

3.4.6 Setting up HTML file format

The AudioGraph Recorder is able to save files in *html* format that actually embeds the real content that is a series of binary objects stored in a *.aep* file. HTML (hypertext markup language) is the text markup language currently used on the World Wide Web. An HTML document is simply a text file that contains the information a user wants to publish in a browser. It contains embedded instructions to indicate how a Web browser should structure or present the document[A11]. Each of the instructions, called elements, is made up of a pair of tags that enclose some textual content, or the other HTML elements such as *embed*. Therefore, the unit required for dealing with an html file can be ported.

Summary

This chapter initially made the comparison between PowerPlant and MFC, and discussed the porting issues of the project. From the discussion, we know that the interface needs to be redeveloped, and however, an interface class in PowerPlant can be mapped to the similar one in MFC. This concept will be used in chapter 5, which contributes to the design and implementation of the AudioGraph Recorder with MFC by choosing proper classes in MFC.

Chapter 4 Comparing a SDK program with a MFC program

MFC is a Windows programming application framework, which encapsulates the Windows APIs to support Windows programming in Object-Oriented way. It provides class libraries and some mechanisms in high levels, which leads to difficulties in programming. As a MFC program is still a Windows program, it requires things such as a program entry function as a C program does, registering at least a window class, and a message loop to retrieve the messages from Windows etc. But MFC buried this inside.

This chapter will compare a native SDK(Software development Kit) Windows program with a MFC program. Some internals of MFC will also be exposed.

4.1 An old fashion Windows program using SDK

The main difference, when comparing a DOS program with Windows, is that the latter is an event-driven environment. This means Windows sits between the hardware and the application. Once Windows detects an event, e.g. a left mouse click, there needs to be some way of letting the application know about the button press so that the application can either process it in a meaningful way or ignore it.

So what is the code needed for a Windows application?

- The application must set up a message loop to extract any message and message handles. The message loop keeps running until the WM_DESTROY message that means the end of the application.
- The application must set up an interface, usually at least a main window, and registered with Windows.
- The application requires initialization and set-up. This step decides how the window shows.

- The application must provide WinMain() function as the entry point of the program, just as regular C program does.

The following program demonstrates the basic code needed for the Windows program, written in C++/SDK (All though the example is written in C++, but it is completely C style to demonstrate a native Windows program.).

The program is a native Windows program to display text in a window. The code is in a *TestWin.cpp* file that has four functions, which are explained below.

4.1.1 The WindowProc() function—The message processing function

The WindowProc() in Listing 4-1 is called by Windows each time a message for the main application window is dispatched. The WindowProc() function is responsible for

```
/********** Listing 4-1.    Block 1- message process function **********/

long WINAPI WindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{ HDC hDC;                  // Display context handle
  PAINTSTRUCT PaintSt;      // Structure defining area to be drawn
  RECT aRect;               // A working rectangle

  switch(message)           // Process selected messages
  { case WM_PAINT:          // Message is to redraw the window
        hDC = BeginPaint(hWnd, &PaintSt); // Prepare to draw the window
        GetClientRect(hWnd, &aRect); // Get upper left and lower right of client area
        SetBkMode(hDC, TRANSPARENT);    // Set text background mode

        DrawText(  // Now draw the text in the window client area
                hDC,            // Device context handle
               "This just a test window?",
               -1,              // Indicate null terminated string
               &aRect,          // Rectangle in which text is to be drawn
               DT_SINGLELINE|   // Text format - single line
               DT_CENTER|       //            - centered in the line
               DT_VCENTER );    //            - line centered in aRect

        EndPaint(hWnd, &PaintSt); // Terminate window redraw operation
        return 0;
    case WM_DESTROY:        // Window is being destroyed
        PostQuitMessage(0);
        return 0;
    default: // call default message processing
        return DefWindowProc(hWnd, message, wParam, lParam);
  }
  return NULL
}
```

analysing what a given message is, and which window it is destined for. It can then call up a whole range of functions, each of which is geared to handle a particular message in the context of the particular window concerned.

The macro WINAPI that is put in front of the function indicates that the function is called by Windows. There are four parameters for the function:

- HWND hWnd – A handle to the window in which the event causing the message occurred.
- UINT message – the message ID, which is a 32-bit value, indicating the type of message.
- WPARAM rParam – a 32-bit value containing additional information depending on what sort message it is.
- LPARAM lParam – a 32-bit value containing additional information depending on what sort message it is.

The above parameters provide information about the particular message causing the function to be called.

Within the function, there is a *switch* statement for the process of decoding messages. For this program, the only message we want to deal with is the WM_PAINNT message, which paints a line of text in the window. The WM_DESTROY message handler results in the call of the API function PostQuitMessage() to generate WM_QUIT message to end the program, as received in the message. The default message handling calls the Window API function DefWindowProc(), which hands the received message back to Windows.

4.1.2 The InitApplication() function—the application-specific initialization

This function defines a WNDCLASS structure variable and then fills up this structure to define what sort window we want to create, and set the WindowProc() function to handle

the received messages for the application. This window is then registered by RegisterClass() to Windows. The code is listed below:

/******** Listing 4-2. Block 2 – application initialization **********/**

```
BOOL InitApplication(HINSTANCE hInstance){

  WNDCLASS WindowClass;     // Structure to hold our window's attributes


  // Redraw the window if the size changes
  WindowClass.style   = CS_HREDRAW | CS_VREDRAW;

  // Define our procedure for message handling
  WindowClass.lpfnWndProc = WindowProc;

  WindowClass.cbClsExtra = 0;   // No extra bytes after the window class
  WindowClass.cbWndExtra = 0;   // structure or the window instance

  WindowClass.hInstance = hInstance;  // Application instance handle

  // Set default application icon
  WindowClass.hIcon = LoadIcon(0, IDI_APPLICATION);

  // Set window cursor to be the standard arrow
  WindowClass.hCursor = LoadCursor(0, IDC_ARROW);

  // Set gray brush for background color
  WindowClass.hbrBackground = (HBRUSH__ *)GetStockObject(GRAY_BRUSH);

  WindowClass.lpszMenuName = 0;  // No menu, so no menu resource name

  WindowClass.lpszClassName = szAppName;  // Set class name

  // Now register our window class
  return(RegisterClass(&WindowClass));
}
```

Lets look at the WNDCLASS structure:

```
Structure WNDCLASS
{
        UINT style;                     //window style
        WNDPROC lpfnWndProc;            //pointer to message processing function
        int cbClsExtra;             // Extra byte after the window class
        int cbWndExtra;             // Extra byte after the window class
        HINSTANCE hInstance;    //the application instance handle
```

```
    HICON hIcon;                    //the application icon
    HCURSOR hCursor;                //the window cursor
    HBRUSH hbrBackground;           //the brush defining the background color
    LPCSTR lpszMenuName; //a pointer to the name of the menu resource
    LPCSTR lpszClassName; //a pointer to the class name
};
```

The WindowClass style can use bit wise OR to combine the options. For example, the *CS_HREDRAW / CS_VREDRAW* indicates to Windows that is to be redrawn if its horizontal width/vertical height is changed.

In the *WNDCLASS*, *cbClsExtra* and *cbWndExtra* allow asking for some extra space internal to Windows for your own use, which is usually zero. The member *hInstance* is used to hold the handle for the current application instance; *hIcon* is used to define the application icon and *hbrBackground* is used to define a Windows object brush to fill an area, which in this case is the client area of the window.

4.1.3 The InitInstance() function—The instance-specific initialization

The function that is listed in listing 4-3 is used to create and display the window. It creates the window by calling the API function CreateWindow(), which specifies window styles, size, position, parent window and menu etc. This window exists after this call, but is not yet displayed on the screen, so another API function ShowWindow() is called. Its first parameter is the window handle returned by the function CreateWindow(), its second parameter decides how the window is to appear on the screen.

The next function called considers the context in the client area, which is not considered to be permanent. So the call API function UpdateWindow() will result in the Windows sending a message to our program requesting that the client area be redrawn.

```
/***************Listing 4-3. Block 3 – instance initialization ****************/

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow) {

  HWND hWnd;                    // Window handle
   hInst= hInstance;

 // Now we can create the window
 hWnd = CreateWindow(
      szAppName,                // the window class name
      "TestWin",   // The window title
      WS_OVERLAPPEDWINDOW,          // Window style as overlapped
      CW_USEDEFAULT,        // Default screen position of upper left
      CW_USEDEFAULT,        // corner of our window as x,y...
      CW_USEDEFAULT,        // Default window size
      CW_USEDEFAULT,        // ....
      0,              // No parent window
      0,              // No menu
      hInstance,          // Program Instance handle
      0               // No window creation data
      );

 ShowWindow(hWnd, nCmdShow);  // Display the window
 UpdateWindow(hWnd);          // Cause window client area to be drawn

  return( TRUE );
}
```

4.1.4 The WinMain() function—The entry point function of the application

```
/********** Listing 4-4. Block 4 – WinMain() function in TestWin.CPP  ********/

#include <windows.h>
HANDLE hInst;  /* current instance */
static char szAppName[] = "TestWin";    // Define window class name

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nCmdShow)
{  MSG msg;                     // Windows message structure
 if(!hPrevInstance){                    /* First instance of App? */
          if(!InitApplication(hInstance))  /* share stuff? */
                  return (FALSE);    /* can not initialize */
  }
 if (!InitInstance ( hInstance,  nCmdShow))
          return( FALSE );

  // The message loop
  while(GetMessage(&msg, 0, 0, 0) == TRUE)  // Get any messages
  {
    TranslateMessage(&msg);           // Translate the message
    DispatchMessage(&msg);            // Dispatch the message
  }

  return msg.wParam;          // End so return to Windows
}
```

This function is the entry point of the program. It has four parameters:

- HINSTANCE hInstance – it is the application instance handle given by the Windows to identify the application at the start of the program.
- HINSTANCE hPrevInstance –it is the handle of the previous instance. In the old days, under Window 3.x, programs shared some common address space, so problems could arise when programs executed simultaneously. Therefore a limit allowing only one program to run at any one time was introduced. Under the 32-bit Windows system however, a program runs in its own address space, so this argument usually is set to NULL.
- LPSTR lpCmdLine – is a pointer to a string, which contains the command line to start a program, such as you clicking Run... items in a menu to run a program.
- int nCmdShow – decides how the window looks (normal or maximized etc).

This function first initializes the application, then initializes the instance, finally setting up the message loop. In the message loop, GetMessage() will be false until retrieved WM_QUIT message from the message queue; TranslateMessage() does the necessary message translation, such as translating the event of the clicking of mouse; DispatchMessage() will direct the message to the proper application, and will cause the Windows to call the WindowProc() function to deal with the message.

In the WinMain(), it has the declaration: MSG msg, which is a structure for messages. Let look at its structure:

```
Struct MSG {
        HWN hWnd;          //handle for the proproriate window
        UINT message;      //the message ID
        WPARAM wParam;  //message parameter ( 32-its)
        LPARAM lParam;  //message parameter ( 32-its)
        DWORD time;      // the time when the message was queued
        Point pt;          //the mouse position
}
```

The two members in the structure, wParam and lParam, decide the exact contents of different kinds message.

When running this application, the result is:

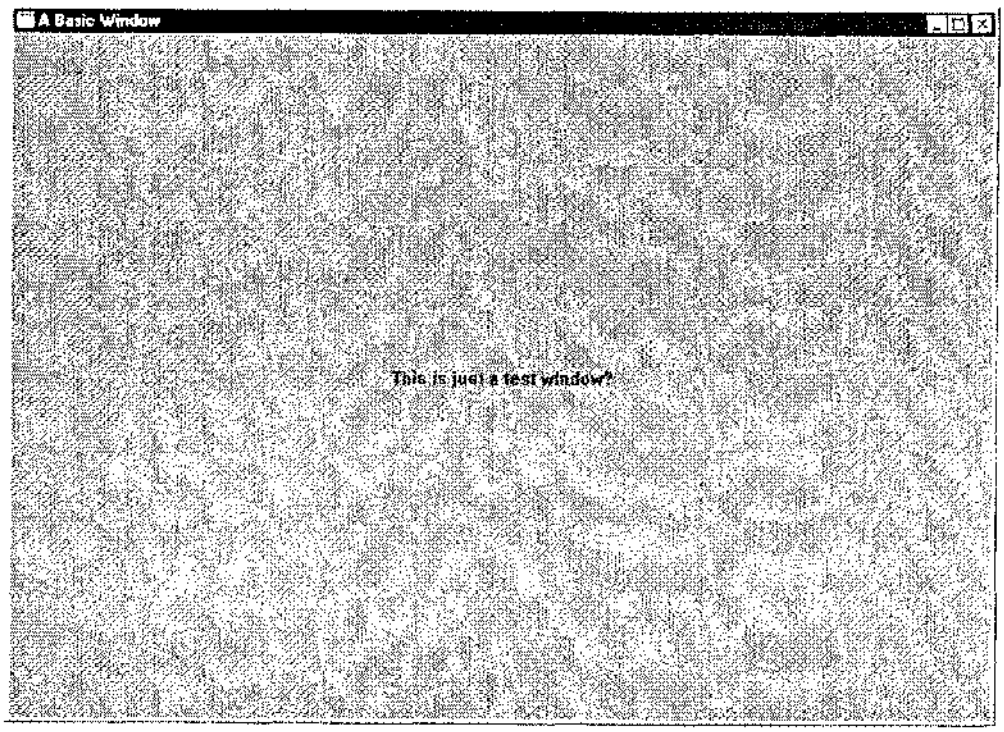


Figure 4-1. A basic window produced by a SDK program

A basic window is obtained on the screen by the code from Listing 4-1 – 4-4, what does the code do?

In the TestWin.cpp, the WinMain() is the entry point of the program. It then calls InitApplication() to perform application-specific initialization and register a window called TestWin with Windows. The WinMain() then calls InitInstance() to create and show a window interface casting from the TestWin window structure mould. Finally, WinMain() goes on to create the application message loop, which continually fetches messages from the message queue, translates and dispatches them to the appropriate window until a WM_QUIT massage is encountered.

Every Windows application must have same basic structure as the TestWin.cpp. MFC is no exception. However, for the sake to getting rid of some boilerplate code, MFC buried some basic code inside that are similar for every Windows application, such as the WinMain() function, the declaration and registration of window classes, and a message loop. Also MFC uses a message mapping mechanism to handle messages in order to prevent the window processing function getting too large as result of too many cases. The next sections will cover some of these issues about MFC's mechanisms, regarding the implementation of a MFC program which produces the same result as the TestWin.cpp program.

4.2 A MFC program

The following is the code of a program in Test.cpp written with C++/MFC, which produces the same result as shown in Figure 4-1.

```
        /**********Listing 4-5.  A MFC program in Test.cpp **************/

#include "afxwin.h"

class CTestWinApp : public CWinApp {   //the application class
  public:
        virtual BOOL InitInstance();
};

class CMainFrame : public CFrameWnd{ //the window class
public:
        CMainFrame(){ Create(NULL, "A Basic Window"); }
        afx_msg void OnPaint();    // ON_WM_PAINT message  handler for redraw the window

        DECLARE_MESSAGE_MAP()
};
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        ON_WM_PAINT()
END_MESSAGE_MAP()

CMainFrame:: OnPaint() {
        CPaintDC dc(this); // device context for painting
        ... //omit. //similar to the  process of WM_PAINT message in WindowProc()
}
BOOL CTestWinApp::InitInstance(){
        CMainFrame* pFrame = new CMainFrame();
        pFrame->ShowWindow(SW_SHOW);
        pFrame->UpdateWindow();
        return TRUE;
};
CTestWinApp  theApp;
```

When Compared to the source code in the TestWin.cpp from Listing 4-1 to 4-4, the size code in Listing 4-5 is reduced by more than 60%. This is one of the benefits of using MFC. However, it is difficult to understand until the internals of the MFC has been mastered.

As mentioned in the last section, a MFC program should have the basic structures in a SDK program. By looking at the above code in the Test.cpp, an InitInstance() function is found, but the entry point WinMain() function for the application is gone, as well as the main event loop and the registration of the CMainframe class. What MFC have done?

In MFC, the entry point function is defined as AfxWinMain() in APPMODULE.CPP of the MFC library, which is part of the framework.

To get the application to work, it firstly defines an only global variable in Test.cpp:

```
CTestWinApp theApp;
```

C++ programs construct their global objects before anything else is done – even before AfxWinMain() is called. Thus, the *theApp* application object is constructed, and eventually AfxWinMain() is executed.

The AfxWinMain() function does quite similar job with the WinMain() function in Listing 4-4, but they are quite different anywhere. The following is a task list which AfxWinMain() does:

1. First, AfxWinMain() calls AfxWinInit() to initialize the framework, which is internal initialization, such as to set the error mode for the application, initialize the application name and path variable within the CWinApp class, and fill CWinApp's help file and profile, and so on;
2. Next, AfxWinMain() calls the application's InitApplication(), whose task is different from that in the SDK program.

With Windows moves to 32 bits system(such as Windows 95 and Windows NT), each application runs in its own address space. Win32 doesn't need to distinguish between the first and subsequent instance. All instance initialization moves to next call InitInstance() in AfxWinMain(). InitApplication() only takes care of the application's document manager used to help manage the MFC document/view architecture.

3. Then AfxWinMain() calls the InitInstance() which is defined in the Test.cpp file of the application. The InitInstance() function initializes an application, sets all the documents for the application if it is a document-based application, and shows the main window.

   In the case of the program in Listing 4-5, InitInstance() only create a main window and show it. UpdateWindow() is then called to update the client area of the main window by sending a WM_PAINT message if the update region is not empty. By the message-mapping mechanism, the OnPaint() message handler for the WM_PAINT message will draw the text on the main window.
4. Finally the Run() function is called to start the generic message loop (GetMessage()... DispatchMessage()).

In a SDK program, the application has to register at least one window with the operating system before the application can display a window. MFC registers window classes during window creation rather than all at once at the beginning of an application, because window classes registration does take time. In the class hierarchy of MFC, Different classes have their own functions where their registration happens, as shown below for some specific classes:

- CWnd::Create()
- CView::PreCreateWindow()
- CFrameWnd:: PreCreateWindow() Or CFrameWnd::LoadFrame()
- CMDIFrameWindow::PreCreateWindow()
- CDialogBar::Create()

By registering classes at window creation, MFC avoids registering window classes that aren't necessary.

4.3 Comparing the SDK program with the MFC program

The SDK program from Listing 4-1 to Listing 4-4 shows the generic structure of native Windows programming. All the Windows programs have to perform many of the same operations to run correctly, and their development steps are usually the same. The MFC program in Listing 4-5 demonstrates a basic Windows application support within MFC. MFC whisks away the necessary boilerplate code, and uses the message mapping mechanism to handle messages in order to prevent the window processing function getting too large as result of too many cases. Keyboard inputting for programmers is a chore process and most bugs are developed during the code inputting stage. MFC does a wonderful job in saving the work needed for programmers to input code.

However, MFC does more than just do away with the necessary boilerplate code, it is a complete application framework to help make programming easier in different ways. Regarding the domain of the multiple document interface of the AudioGraph Recorder, MFC provides some great architecture to deal with the dynamic creation of the MDI child window and data persistence. Understanding these architectures is essential to the design and implementation of the AudioGraph Recorder on PC.

Next, we will discuss some internals of MFC about run-time class information, dynamic creation, serialisation support and document/view architecture in MFC using the class objects in the project as example.

4.4 Run-time class information (RTCI)

In MFC, one of the common techniques is the use of macros (DELARE / IMPLEMENT) to add various features to the application classes. In the beginning of the definition file MainFrm.h and the implementation file MainFram.cpp of the CMainFrame class of this

application (CMainFrame is for the main frame object in this project interface), the following macros are added:

```
DECLARE_DYNAMIC(CMainFrame) /
IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd)
```

The above macros add the CObject's run-time class information (RTCI) feature to the CMainFrame class, so that the information about the CMainFrame class object such as class name and parent can be determined at run time.

The current C++ language also supports Run-Time Type Information (RTTI) of an object. RTTI is a mechanism that allows the type of an object to be determined during program execution, which is similar to RTCI. However, MFC does not use the C++ RTTI. This is due to the development history of MFC, in which the MFC development team was separated from the C++ language development team. In the very early days of the MFC development, the MFC team needed the information about objects at run time, but this feature was not implemented in C++ by the language team at that time. So the MFC team implemented their own support of RTCI.

The implementation of the Macintosh AudioGraph Recorder used the RTTI feature in C++ to do an object type safe check and an object dynamic cast for polymorphic types.

How does MFC implement RCTI? The CMainFrame class is used as an example to explain it.

After running the pre-processor that translates all the defined macros, the macro: DECLARE_DYNAMIC(CMainFrame) in MainFrm.h will become:

```
protected:      static CRuntimeClass* PASCAL _GetBaseClass();
public:         static const AFX_DATA CRuntimeClass classCMainFrame;
                virtual CRuntimeClass* GetRuntimeClass() const;
```

The code adds the declaration of a member and two member functions to the main frame class:

- A static member of the *CRuntimeClass* type called *classCMainFrame*. The *CRuntimeClass* is a structure used to hold the class run-time information.
- A static function *_GetBaseClass()* which returns the run-time class member of the base class of CMainFrame;
- Another funtion *GetRuntimeClass()* which returns the correct run-time class information of *classCMainFrame*.

After running the preprocessor, the IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd) macro is translated into:

```
CRuntimeClass* PASCAL CMainFrame::_GetBaseClass()
        { return &CMDIFrameWnd::classCMDIFrameWnd }
AFX_COMDAT const AFX_DATADEF CRuntimeClass CMainFrame::classCMainFrame =
{
        "CMainFrame", sizeof(CMainFrame), 0xFFFF, NULL,
        &CMainFrame::_GetBaseClass, NULL
};
CRuntimeClass* class_name::GetRuntimeClass() const
        { return &CMainFrame::classCMainFrame; }
```

The above code is straight forward. It fills a run-time class structure and implements the function declared by the DELARE_DYNAMIC macro. Let look at the initialization of *CMainFrame:: classCMainFrame* how to fill the following members of the *CRuntimeClass* structure:

1. The class name for the run-time class: m_lpszClassName = "CMainFrame".
2. The size of the CMainFrame object: m_nObjectSize = sizeof(CMainFrame).
3. The scheme number of loaded class: m_wSchema = 0xFFFF.

4. The member of *CRuntimeClass*: m_pfnCreateObject = NULL.
5. The run-time class of its base class: m_pBaseClass = CMDIFrameWnd::classCMDIFrameWnd.

The macro *DECLARE_DYNAMIC/ IMPLEMENT_DYNAMIC* set in the main frame class creates the class information at run-time. It is useful for a safety check, such as the *CObject::IsKindOf* function allows to do a type-safe cast down to a derived class. The following is an example used in the project:

```
CSlideFrmView:: OnInitialUpdate() {
        CSlideDoc* pDoc;
        pDoc = GetDocument();
        if ( pDoc -> IsKindOf( RUNTIME_CLASS (CSlideDoc)
                SetScrollSizes(MM_TEXT,pDoc->GetDocSize() );
}
```

In this function, after a pointer of a current active slide document object is achieved, the *IsKindOf function* is used to make sure the pointer is pointed to the type of the slide document in this application.

The run-time class information is also used by the dynamic creation of an object.

4.5 Dynamic Creation

Dynamic creation is the process of creating an object of a specific class at run time, which is also one of the functions supported by CObject. In the dynamic creation of an object, the object is created by the *CreateObject* member function of *CRuntimeClass*.

As this application is an MDI application, the Lecture Window and the Slide Window as MDI child windows are dynamically created at run time. Therefor the document, view, and frame class for the Lecture Window and the Slide Window should support dynamic

creation. To support dynamic creation, the document class, the view class and the MDI child frame class of the application have the macros:

```
DECLARE_ DYNCREATE (class_name) /
IMPLEMENT_DYNCREATE(class_name, base_class_name)
```

The DECLARE_DYNCREATE macro output differs from the DECLARE_DYNAMIC macro by only adding one statement:

```
static CObject* PASCAL CreateObject();
```

In addition to the IMPLEMENT_DYNAMIC macro, IMPLEMENT_DYNCREATE generates the CreateObject() member function:

```
CObject* PASCAL class_name::CreateObject()
        { return new class_name; }
```

In the CreateObject function the *new* operator attempts to dynamically allocate (at run time) an object of the *class_name* class, and yields a pointer to that object. Then the IMPLEMENT_DYNAMIC macro is invoked by passing this pointer to the *CRuntimeClass::m_fnCreateObject* (Remember in the run-time class implementation, this member *m_fnCreateObject* is set to *NULL*).

At the heart of MFC dynamic creation, is the CRuntimeClass::CreateObject() function. Its pseudocode is given below:

```
CObject* CRuntimeClass::CreateObject()
{       if (m_pfnCreateObject == NULL)   return NULL;
        CObject* pObject = NULL;
        TRY{    pObject = (*m_pfnCreateObject)();
        }END_TRY
         return pObject;
}
```

The CreateObject function first checks that the *m_pfnCreateObject* is equal to *NULL* or not, checking if the correct macro being used. If the *IMPEMENT_ DYNAMIC* macro is used, the m_pfnCreateObject will be not initialized. Then the pointer stored in the *CRuntimeClass:: m_pfnCreateObject* is passed to *CRuntimeClass:: CreateObject()*.

Compared to the C++ language, the dynamic creation of an object is no mystery, as it just use the *new* operator in C++ to create a new object, but it uses the RTCI of an object for safe type check. The following is an example for using the dynamic creation of a CObject-derived CAobject class object:

```
CRuntimeClass* pRuntimeClass = RUNTIME_CLASS(CAobject);
CObject*  pObject = pRuntimeClass -> CreateObject();
ASSERT (pObject ->IsKindOf( RUNTIME_CLASS( CAobject ) ) );
```

In this example, a new CAobject object is created by the CreateObject function of the run-time class member of Caobject before using IsKindOf function to do the safe check. The ASSERT macro is defined in MFC to evaluate its argument. If the result is 0, the macro prints a diagnostic message and aborts the program

4.6 Serialization

As mentioned in chapter 3, MFC uses a serialization mechanism to support persistence. "Serialization" is the process of writing or reading the contents of an object to and from a file. MFC uses an object of the *CArchive* class as an intermediary between the object to be serialized and the storage medium. A *CArchive* object provides a type-safe buffering mechanism for writing or reading serializable objects to or from a *CFile* object. In particular, it has overloaded insertion (<<) and extraction (>>) operators to support both primitive types and CObject-derived classes for writing and reading operations.

To support serialization, the *DECLARE_SERIAL / IMPLEMENT_SERIAL* macros must be added in a CObject-derived class. Compared to the *DECLARE_DYNAMIC macro*,

DECLARE_SERIAL has one additional declaration for a global extraction operation as a friend to the CObject-derived class, which is:

```
AFX_API friend CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb);
```

The macro *IMPLEMENT_SERIAL(class_name, base_class_name, wSchema)* has one additional parameter *wSchema* when compared to the macro for RTCI and dynamic creation. *WSchema* is used to pass the scheme number to the class's *CRuntimeClass* member: *m_WSchema* to determine if the correct macro is used by the function *CObject::IsSerializable()*, which code is listed below:

```
BOOL CObject::IsSerializable() const{
        returen (GetRunTimeClass()->wSchema != 0xFFFF;
}
```

Remember: *m_ Wschema* = 0xFFFF in the cases of RTCI and dynamic creation. So if the serialization macros is correctly used in a class, the *IsSerializable* function returns *true*.

IMPLEMENT_SERIAL also generates the implementation of the extraction operation in addition to *IMPLEMENT_DYNAMIC* as:

```
CArchive& AFXAPI operator>>(CArchive& ar, class_name* &pOb) {
        pOb = (class_name*) ar.ReadObject(RUNTIME_CLASS(class_name));
        return ar;
}
```

The macro *RUNTIME_CLASS* returns the static CRuntimeClass member data of the class. *IMPLEMENT_SERIAL* overloads the operator ">>" for a class (*class_name*) so that *CArchive:: ReadObject()* is called. The *CArchive::ReadObject()* fulfils the following tasks:

1. *ReadObject()* first reads the CRuntimeClass structure from the file, and assign it to a local variable;
2. Then it uses the CRuntimeClass::CreateObject() to allocate a new object based on the class just acquired;
3. It then calls the object's *Serialize()* function to get the object data from the file to initialize its data member.
4. Finally the object is returned to the *CArchive:: ReadObject()* function.

The insertion operator (<<) for objects is defined as a global variable in MFC. It calls the *CArchive::WriteObject()* to write out the CRuntimeClass information. Then *WriteObject()* calls the object's *Serialize()* member function to serialize its data members.

Serialization for an appliction starts by clicking **File|Save** or **File|Open...** menu items. A message then is sent to the document class in the application to cause the message handler(*CDocument::OnSaveDocument() / CDocument::OnOpenDocument()*) to handle the message. The handles construct a *CArchive* locally and attaches it to a file opened by a common file dialog. Next the handler calls the document's *Serialize()* member function to get involved with the overloaded insertion/extraction operator.

4.7 Document / view architecture

Managing different types of data for an application is a difficult task. The MFC document/view architecture provides a single, consistent way of co-ordinating the application's data.

MFC applications use a programming model that separates a program's data from the display of that data and from most user interaction with the data. In this model, an MFC document object reads and writes data to persistent storage. A separate view object manages the data display, from rendering the data in a window for a user to select and edit the data. The view obtains the data to be displayed from the document and communicates back to the document when any data changes.

In MFC, the CDocument class handles data management and the CView class handles the user-interface management. The document is just a place to keep data, and a view is a place to represent that data. The MFC document/view architecture provides the basic structure for managing documents and views. It has four components that work together:

1. *CDocument*—an object used to store or control data.

   A *CDocument* is owned by a *CDocTemplate*. Documents have a list of currently open views (a *CPtrList*). Documents do not create/destroy the views, but they are attached to each other after they are created. The *CDocument* is a friend of *CView*.

2. *CView*—an object used to display a document's data and manage user interaction with the data.

3. *CFrameWnd*(or one of its variations)—an object that provides the frame around one or more views of a document.

   The *CFrameWnd* is responsible for creating windows in the client area of the frame. The *CWinApp* application object owns all frame windows in the application. For multiple document interface (MDI), the *CMDIFrameWnd* class serves as the main application frame window that contains all the child frame. Each child window is of class *CMDIChildWnd*.

4. *CDocTemplate* (or *CSingleDocTemplate* or *CMultiDocTemplate*)— an object that co-ordinates one or more existing documents of a given type and manages creating the correct document, view, and frame window objects for that type.

   The *CDocTemplate* is the creator and manager of documents. It owns the documents which it creates. For an MDI application, the class

CMultiDocTemplate keeps a list (a *CPtrList*) of all the currently open documents created from that template. *CDocTemplate* is a friend of CDocument.

The document template objects are created by the application object (derived from *CWinApp*). There is only one *CWinApp* object in an application. The application object owns a list of document templates (a *CPtrList*). An application may have one or more document type.

The diagram in Figure 4-2 shows the object interrelationships in the document/view architecture:

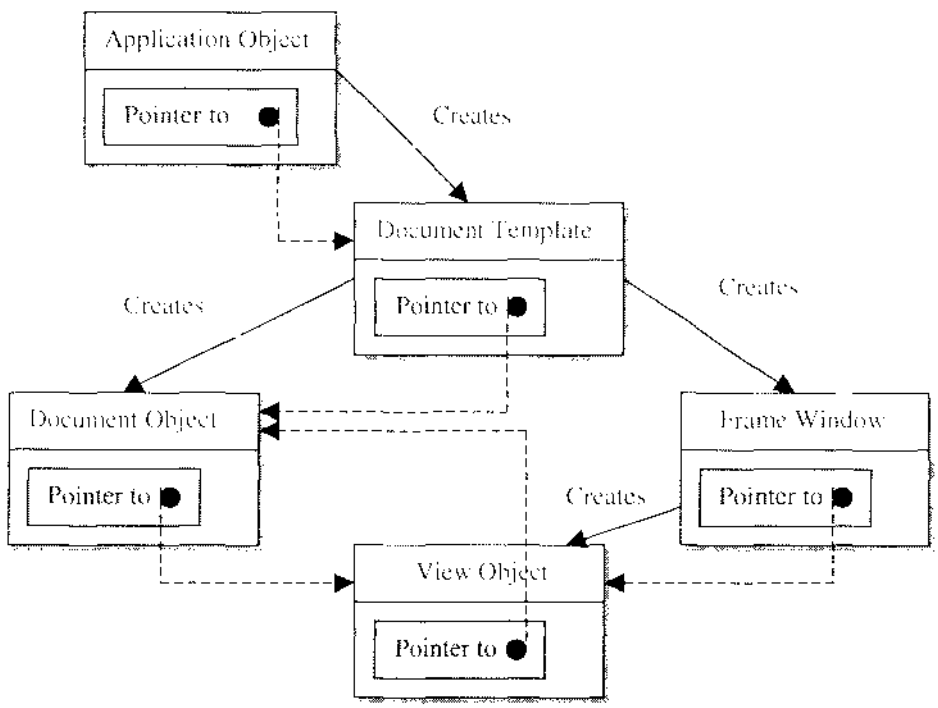


Figure 4-2. The object interrelationship in the document/view architecture

The diagram indicates the object creating sequence and the object relationships. The dashed arrows in the diagram show how pointers are used to indicate that an object is related to another.

Let's follow the creation of a new document in the AudioGraph Recorder application to see how the document/view architecture works.

In the implement file *AudioGraph.cpp* of the application class *CAudioGraphApp*, there is a massage map entry:

```
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
```

This is a command massage generated by clicking *File|New* menu items. *ID_FILE_NEW* is the ID for the menu items. *OnFileNew* is the message handler.

If there are more than one document templates stored in the template list, *CWinApp::OnFileNew* results in the creation of a modal dialog to let the user choose the type of new document to create. Then *CMDIDocTemplate:: OpenDocumentFile()* is called .

CMDIDocTemplate::OpenDocumentFile() does some important jobs. First, it calls *CDocTemplate::CreateNewDocument()* to create a new document, based on the run-time information of the document class stored in the document template. Then, *CMDIDocTemplate::OpenDocumentFile()* calls the *CDocTemplate::CreateNewFrame()* to create a MDI child frame object, based on the run-time information of the MDI child frame which is stored in the document template.

The MDI child frame is responsible for its attached view creation. So the creation of the MDI child frame results in a call of CFrameWnd::CreateView() to create a new view in its client area based on the view run-time information stored in the document template.

At this stage, the document, MDI child frame and view are created. The flow of the above process for a *File|New* operation is shown in Figure 3-2 :

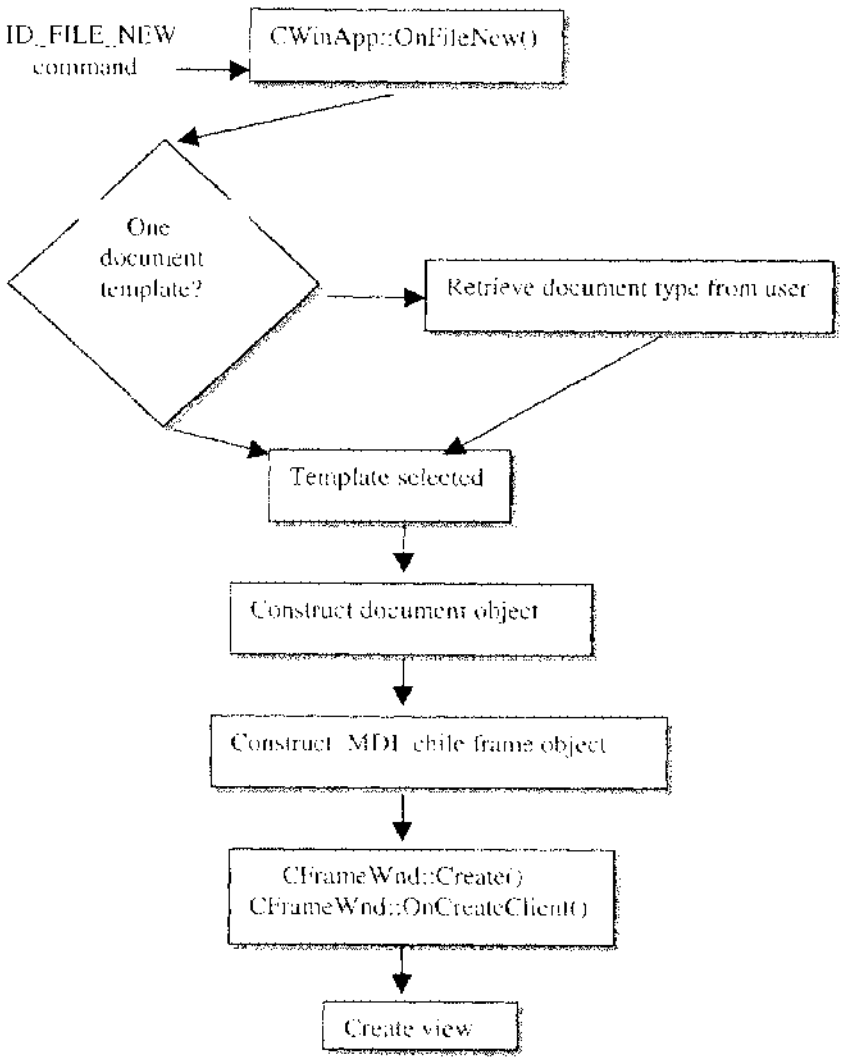


Figure 3-2 The flow control for the File|New operation

In the AudioGraph Recorder application, all the code involved with the document/view architecture is in the *CAudioGraphApp::InitInstance()* function, which is listed below:

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
        IDR_AUDIOGTYPE,
        RUNTIME_CLASS(CAudioGraphDoc),
        RUNTIME_CLASS(CChildLectureFrm),
        RUNTIME_CLASS(CLectureFrmView));
AddDocTemplate(pDocTemplate)
```

In the code first constructs a document template by the information passed to it by its parameters: the resource (specified by the resource IDR_AUDIOGTYPE), the run-time information of the document, MDI child frame and view class. IDR_AUDIOGTYPE is the resource ID for the icon, menu and toolbar that relate to the MDI child frame. The document template is then added in the template list of the application object in the code.

We now know that the process of the document/view architecture in an application starts by receiving an ID_FILE_NEW command. This command is generated by clicking the File|New menu options. But when a MDI application, such as this application starts to run, a MDI child window is automatically opened. At the start of the program, the File|New menu option has no chance to be clicked, so what happens? The answer is in the code following the construction of the document template in the *CAudioGraphApp:: InitInstance()* function:

```
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);
// Dispatch commands specified on the command line
if (!ProcessShellCommand(cmdInfo))
        return FALSE;
```

In the above code, a *CCommandLineInfo* object is constructed first. The *CCommandLineInfo* class aids in parsing the command line when an application starts. It contains a member, which is defined as:

```
enum { FileNew, FileOpen, FilePrint, FilePrintTo, FileDDE,
        AppUnregister, FileNothing = -1 } m_nShellCommand;
```

The default value for *m_nShellCommand* is set to *FileNew*. When processing *ProcessShellCommand() in CAudioGraphApp::InitInstance()*, after it distinguishes the *FileNew* value, it calls the *CWinApp::OnFileNew()* function. This starts the process given in Figure 3-2 to create the MDI child frame and document.

Summary

After compared a SDK program with a MFC program, we have discussed about the run-time class information, dynamic creation, serialization and document/view architecture, based on the AudioGraph Recorder application. These techniques is essential to MFC programming. It is applied to the interface design and implementation of the application in Chapter 5.

Chapter 5 The Design and implementation of AudioGraph Recorder

This chapter describes the design and implementation of AudioGraph Recorder on PC using MFC. Firstly, a brief explanation is given about the achievements of the PC AudiGraph Recorder project, as well as some major problems that have occurred during the process of the design and implementation. Then the design is described by looking the MFC class hierarchy and choosing the proper classes for the interface objects. A modification is made to the class diagram specified in chapter 2. Details of the implementation of some of the interface classes will be discussed in order to demonstrate how the problems have been solved. Finally, examples are given also of the implementation of some of the application's functionality, such as drawing etc. to show how to port some code from the Macintosh AudioGraph Recorder to the PC.

5.1 The achieved result

The interface achieved is complete and is fully consistent with the Macintosh version. The detail of the work completed is given in the following list:

- All the windows are implemented, including the main window, the Lecture Window, the Slide Window, the tool windows (the Tool Window, the Attribute Window, and The Edit Console Window).
- All dialogs are implemented, such as the Preference dialog, the Rename dialog, the Pause dialog, the Recording dialog and etc.,
- The main menu is constructed and implemented, as well as the two pop up menu (the Pen Colour menu, the Pen Width menu) related to the Tool Window.
- The tool bar in the Edit Console is implemented, which is correspondingly related to some menu options of the **Edit** menu item in the main menu.
- The state changes of the controls in the Lecture Window, the Edit Control, the Tool Window, the Slide Window, and the Attribute Window are implemented.

- The change in the states of the menu items and options, the text in the menu options are implemented.
- The drawing of rectangle, line, circle with different colour and width, and inserting text in the Slide window are properly implemented.
- The storing and retrieving of the data in each individual Slide window to and from a file is implemented.

The interface has been tested in all of the currently available Windows operating systems and acts just like the Macintosh version.

5.2 Problems encountered

During the design and implementation of the AudiGraph Recorder interface, a number of problems have been encountered and solved. The following lists some of the major problems:

- The Lecture Window is actually a dialog-style, but it needs to connect with a document. To take full advantage of the document/view architecture of MFC for the dynamic creation of multiple windows and to bind this to the document, it needs to implement a view object in the Slide Window. So all the control will appear on the view. In the Macintosh version, a tab control was used in the Lecture Window. MFC has a property sheet class for fully functional tab control, which is chosen for the control of the title view and the thumbnail view in the Lecture window. The property sheet object needs to be created and properly positioned with the other button and controls in the Lecture Window, and the buttons must visually appear on the property sheet object.
- There is a list control object in the property sheet of the Lecture Window. The MFC list control fails to insert a separation line for each columns and row. In PowerPlant, there is a LTable object to provide a fully functional list control. So some functionality needs to be added to the list control class of MFC, and a icon needs to be put on the header of the list control.

- The Slide Window is actually the working space, which is created using the document/view architecture of MFC. So the button controls will be in the view object of the Slide Window with scroll bars. In Windows, the default horizontal scroll bar of views has the same width as the view. MFC fails to provide the interface function to access the default scrollbar. To set the button controls along with a horizontal scrollbar, a scrollbar object needs to be created to replace the default horizontal bar in the Slide Window view, and to be set up properly to scroll the view.
- The Tool Window and the Attribute Window are actually tools windows, with functionally just like a toolbars in Windows. The messages generated by the controls in the two tool windows need to be commands so that the objects in the command hierarchy can have a chance to handle them. But MFC fails to support the initialization of the bitmap on bitmap buttons in its dialog bar class. So suitable procedures need to be added in the class, so that the bitmap button controls can be properly initialized before the Tool window and the Attribute are shown on the screen.
- The Edit Console acts as a tool window also. However, it has icons to be inserted dynamically and a toolbar, and the icons can be scrolled. Therefore the tool bar class and its derivatives are not suitable for the Edit Console object. The Edit Console needs to have a scroll view and is implemented as a pop window. As the icons in the Edit Console are dynamically created, their IDs are assigned dynamically, so the Edit Console class can not set up a handle for individual mouse-click messages sent by each icon button. As a result, the Edit Console class needs to set up a handler properly for all the messages by its contained icons and set the states of the icons correspondingly.
- The scroll view objects in the Slide Window and in the Edit Console, require dynamic size for their contents.

Due to the consistency of the interface and its behaviour with the Macintosh version, some of the problems were raised above due to some of the features and behaviours not being directly supported by MFC. This results in adding some functionality to some classes in MFC and changing some default processes of the function calling in sequence.

In the following sections, some major windows implementation will be discussed in detail, in the mean time the solutions will be exposed.

5.3 The design of the interface

The interface class diagram of the PC version AudioGraph Recorder is shown below:

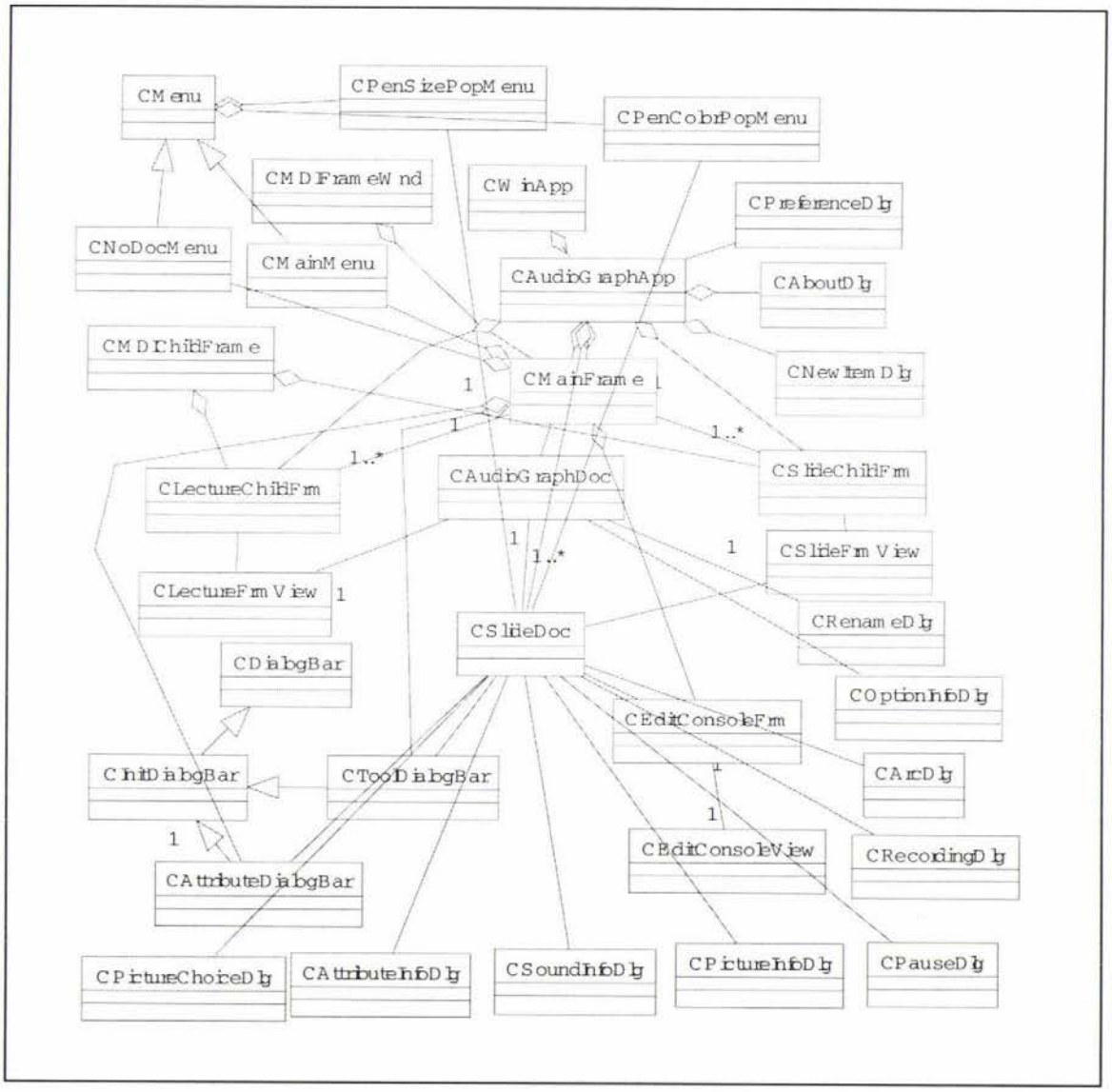


Figure 5-1. The interface Class Diagram

Compared to the interface class diagram of the Macintosh version, some modifications have been made in the interface class diagram of the PC version, which are listed below:

- A CNoDocMenu class is added in the application, which appeares when none of documents are opened. In the Macintosh version application, most of menu options in the main menu are disabled except the ***File|New Lecture*** in the case of no document existing in the application. In order to save the code to disable the menu options in the main menu, a CNoDocMenu class menu object is loaded when all the child windows are closed in the application.
- A CMainFrame class for the main window is added in the application, as a Windows application must have a main window.
- Compared to the Macintosh version, the PC version application has two document classes (CAudioGraphDoc for a lecture document and CSlideDoc for the slide document). This is because the types of data in the Slide Window and the Lecture Window are different, and the MDI Lecture Windows and the Slide Windows are created using document/view architecture of MFC in this application, in which each of these windows relates to a corresponding document.

  In order to save a lecture data in a file as the Macintosh version application does, the CAudioGraphDoc holds a pointer list of CSlideDoc objects which only relates to the lecture.
- In the class diagram, the CMainFrame class has a "Whole-Part" relation with the two menu classes and the three tool windows, as it is the responsibility of the main frame object to create them, as well as a "Association" relation with the Slide Window class and the Lecture Window class because the main frame is their parent.

Proper classes in MFC are chosen for the interface components to be derived from. Some major objects class are listed below:

- CMainFrame—derived from CMDIFrameWnd for the main frame. The CMDIFrameWnd provides the functionality of a Windows multiple document interface (MDI) frame window.
- CLectureChildFrm—derived from CMDIChildWnd for the Lecture Window frame. The CMDIChildWnd class provides the functionality of a Windows multiple document interface (MDI) child window.

- CSlideChildFrm—derived from CMDIChildWnd for the Slide Window frame.
- CLectureFrmView—derived from CFormView for the view object of the Lecture window. A form view is essentially a view that contains controls.
- CSlideFrmView—derived from CScrollView for the view object of the Slide Window to support the scrolling ability.
- CInitDialogBar—derived from CDialogBar that provides the functionality of a Windows dialog box in a control bar. The messages sent by the controls in the dialog bar are commands. CInitDialogBar extends the ability of the CDialogBar to handle the bitmap buttons in the CinitDialogBar object.
- CAttributeDialogBar—derived from CinitDialogBar for the Attribute Window.
- CToolDialogBar—derived from CInitDialogBar for the Tool Window.
- CEditConsole—derived from CMiniFrameWnd for the frame of the Edit console. A CMiniFrameWnd object represents a half-height frame window typically seen around floating toolbars. The Edit console is designed to be a pop window, so the CEditConsole object has a pop-window style.
- CEditConsoleView—derived from CScrollView for the view object in the Edit Console to support the scrolling ability.

The object diagrams are not given here as they are similar to the ones given in chapter 2. As discussed earlier, an interface component from the PowerPlant visual class hierarchy can simply be mapped into one component from the MFC visual hierarchy. In general, the following rules are applied to the modification in the object diagrams given in chapter two for the design and implemented of the objects in the interface:

- Most standard controls in PowerPlant can be directly mapped to the corresponding controls in MFC, such radio box, check box, static text, progress bar, slide bar, push button and group box etc. Buttons with graphics on such as bevel button in PowerPlant can be mapped to the bitmap button in MFC instead.
- An object of the LDialog class in PowerPlant can be replaced by a CDialog object in MFC.

- An Object of LWindow can be replace by an object of CFrameWnd or its derivatives in MFC.
- For animation control, the chasing arrow object can be replace by a CAnimateCtrl object in MFC.
- A tab control object can be replaced by a CPropertySheet object. And the multipe page object can be replaced by a CPropertyPage object in MFC.

The application used the MFC library as a dynamic-link library (DLL) to reduce the size of the application, and to save the disk and space when running the application[A3].

> A dynamic-link library (DLL) is an executable file that acts as a shared library of functions. Dynamic linking provides a way for a process to call a function that is not part of its executable code. The executable code for the function is located in a DLL, which contains one or more functions that are compiled, linked, and stored separately from the processes that use them. DLLs also facilitate the sharing of data and resources. Multiple applications can simultaneously access the contents of a single copy of a DLL in memory.
>
> Dynamic linking differs from static linking in that it allows an executable module (either a .DLL or .EXE file) to include only the information needed at run time to locate the executable code for a DLL function. In static linking, the linker gets all the referenced functions from the static link library and places it with an application code into the application's executable code. Using DLLs instead of static link libraries makes the size of the executable file smaller. If several applications use the same DLL, this can be a big saving in disk space and memory.

The following sections will discuss the implementation of some specific windows in the AudioGraph Recorder.

5.4 implementation

5.4.1 The Lecture Window

The Lecture window is created using the document/view architecture. It has a CFormView-derived CLectureFrmView object for its view, which is a dialog-style form view.

To implement a property sheet in the form view, the following classes need to be implemented:

- *CLectureViewPropertySheet*—derived from *CPropertySheet* for the tab control in the form view.
- *CListPage*—derived from *CPropertyPage* for the list page to display slide names in the tab control.
- *CThumbnailPage*—derived from *CPropertyPage* for the thumbnail page to display the thumbnail image in the tab control.
- *CSlideNameListCtrl*—derived from *CListCtrl* for a list box inserted into the list page. As the CListCtrl class doesn't have the functionality to insert separation line for each column and row for a list view in the report mode (In this mode, the list view has a header on its top), the *CSlideNameListCtrl* class performs this additional funtionality to CListCtrl class.

Because the property sheet can't be constructed in the dialog resource of the form view by the resource editor like other button controls, we need to put a picture holder object in the dialog resource of the form view, which is used to decide the position and size of the property sheet when it is created. To let the right side border of the property sheet disappear when the Lecture Window appears on the screen, the solution is to make the dialog resource of the list page and the thumbnail object wider than the picture holder object. And we put a picture object associated with a bitmap along the proper height of the picture holder object above the button controls in the dialog resource of the form

view. The bitmap is in white with one pixel height. As the color of the top border of the property sheet object is white, the button control will visually appear on the property sheet.

Then we will implement the classes for the Lecture Window.

The CSlideNameListCtrl class

The list view control does not provide any visual separation line between the columns and the rows. The solution is drawing the separation lines in the list control view. To do any own drawing, we need to override the inherited *OnPaint()* function from *CListCtrl* in the class. The framework calls this member function of the *CSlideNameListCtrl* class every time it's necessary to repaint the window, such as when inserting a item in the list control or resizing the Lecture Window. The code for the OnPaint function are listed in the Listing 5-1 of next page:

```
/***********Listing 5-1 ***********/
Void CSlideNameListCtr::OnPaint()
{       CListCtrl::OnPaint();

        int nItemCount= GetItemCount(); //get the item number in list box
        int nRowHeight;  //the height of each row
// Paint vertical and horizontal lines// Draw the lines only for LVS_REPORT mode
    if( (GetStyle() & LVS_TYPEMASK) == LVS_REPORT )
    {  // create a pen for drawing
     CClientDC dc(this );
     CPen pen(PS_SOLID,1,RGB(192,192,192));
     CPen *oldpen = dc.SelectObject(&pen);

     RECT rect;
        //get the height of each row
    if (GetItemRect( 0, &rect, LVIR_BOUNDS))  nRowHeight = rect.bottom - rect.top;

     CHeaderCtrl* pHeader = (CHeaderCtrl*)GetDlgItem(0);
     int nColumnCount = pHeader->GetItemCount();  //how many columns

     // The bottom of the header corresponds to the top of the line
     pHeader->GetClientRect(&rect);
     top = rect.bottom;
    // Now get the client rect so we know the line length and when to stop
     GetClientRect(&rect);

    if (VerticalGrid)
    {// The border of the column is offset by the horz scroll
       int borderx = 0 - GetScrollPos(SB_HORZ);
       for (int i = 0; i < nColumnCount; i++)
       {// Get the next border
          borderx += GetColumnWidth(i);
          //if next border is outside client area, break out
          if ( borderx >= rect.right )                break;
           // Draw the line
         dc.MoveTo(borderx-1,top);
         dc.LineTo(borderx-1,top+height*nItemCount);// rect.bottom );
         }
      }
 // Draw the horizontal grid lines
        if (HorizontalGrid) {
                GetClientRect(&rect);
                int width = rect.right;
                for (int i = 1; i <= nItemCount;i++)//GetGetCountPerPage(); i++)
                {       dc.MoveTo(0, top + height*i);
                        dc.LineTo(width, top + height*i);
                 }
         }
  dc.SelectObject(oldpen); //rstore the old pen
   }
}
```

OnPaint() firstly create a device context object and a pen object which will be selected in the device context for drawing. A device context is a data structure that contains

information to allow Windows to translate output requests. MFC defined some device context classes, such as CClientDC for the drawing on a client area.

After calculating the start points and end points for the horizontal and vertical separation lines using the position and size of the header and the client of the list view control, OnPaint calls the *MoveTo()* and *LineTo*() functions of the *CClientDC* deveice context object to draw the vertical and horizontal lines in the list view control.

When inserting an item in the list view control, the two separation lines must be drawn. The *CListCtrl::InsertItem()* function for inserting an item in the list control needs to be overriden as follows:

```
/************Listing 5-2 ************/

int CSlideNameListCtrl::InsertItem(int nItem, LPCTSTR lpszItem)
{
    CListCtrl::InsertItem( nItem, lpszItem );

    DrawVerticalGrid(true);
    DrawHorizontalGrid(true);

    return nItem;
}
```

The *InsertItem()* function sets the flags to true for drawing vertical and horizontal lines in addition to the default function of *CListCtrl* class. When this function is called, the *OnPaint()* will be called for redrawing the list view control. As the flags for drawing vertical and horizontal lines are set to true, the separation lines will be drawn.

The CListPage class

This class does some initialization jobs for the list view control which is put into the dialog resource of the *CListPage* class. Because the list control view needs to put an icon on its header and change the default font for the header caption. We need to override the member function *OnInitDialog()* of the *CPropertyPage* class that is inherited by the *CListPage* class.

This member function *OnInitDialog()* is called in response to the *WM_INITDIALOG* message. This message is sent to the object during the *Create, CreateIndirect, or DoModal* calls that result in the creation of the object, which occurs immediately before the dialog box is displayed. The code for the *OnInitDialog* function is listed below:

```
/***********Listing 5-3 ***********/

BOOL CListPage::OnInitDialog()
{
        CPropertyPage::OnInitDialog();
//set two coloums and set the text on the header
        m_ListCtrl.InsertColumn(0," Episode",LVCFMT_CENTER,330,-1);
        m_ListCtrl.InsertColumn(1,"Duration",LVCFMT_CENTER,81,-1);
//get the header of the control
        CHeaderCtrl* pHeader = m_ListCtrl.GetHeaderCtrl();
//create an font object
        m_FontEdit->CreateFont( -10, 0, 0, 0, 900, 0, 0, 0, 1,
                        0, 0, 0, 0, _T("Arial") );
        pHeade->SetFont( m_fontEdit ); //set font for the header
//set the resouce icon to the image list
        m_pHdrImages->Create(5, 15, ILC_MASK, 2, 2);
        m_pHdrImages->Add(AfxGetApp()->LoadIcon(IDI_HEADER));

        pHeade->SetImageList(m_pHdrImages);
        HDITEM    curItem;
//set the icon in the image list m_pHdrImages on the first column of the header
        pHeade->SetImageList(m_pHdrImages);
        pHeade->GetItem(0, &curItem);
        curItem.mask= HDI_IMAGE | HDI_FORMAT;
        curItem.iImage= 0;
        curItem.fmt= HDF_LEFT | HDF_IMAGE | HDF_STRING;
        pHeade->SetItem(0, &curItem);

        return TRUE;
}
```

In the above code, *m_pHdrImages* is declared as a private member of the *CImageList* type in the CListPage class. An "image list" is a collection of same-sized images, each of which can be referred to by its zero-based index. And *m_ListCtrl* is the public member of the *CSlideNameListCtr* type.

The *OnInitDialog()* function first sets up two columns in specified width with captions for the list control. And set the proper font for the header text of the list control. After

getting a icon from a resource (ID is IDI_HEADER), finally the *OnInitDialog()* function sets the icon to the first column of the header of the list control.

The CLectureViewPropertySheet class

This property sheet has two page objects, so in the class we need to declare as follows:

```
public:
        CListPage m_ListPage;
        CThumbailPage m_ThumbnailPage;
```

To put this two page objects in the property sheet, we need to add the following codes in the constructor of the class:

```
AddPage(&m_ThumbnailPage);
AddPage(&m_ListPage);
```

The *AddPage()* function adds the page object on its reference parameter with the rightmost tab in the property sheet. So the default page is the list page that shows a list view.

The CLectureFrmView class

This class is responsible for the creation of the property sheet. To create the property sheet in the form view, we override the member function *OnInitialUpdate()* which inherited from its base class *CFormView*. The code is listed in Listing 5-4.

```
/************Listing 5-4 ************/

void CLectureFrmView::OnInitialUpdate()
{
        CFormView::OnInitialUpdate();
        ResizeParentToFit();

// create and asociated the property sheet with the "place holder"
        CWnd* pwndPropSheetHolder = GetDlgItem(IDC_PLACEHOLDER);

        m_pFormViewPropSheet = new CLectureFrmView (pwndPropSheetHolder);

        if (!m_pFormViewPropSheet ->Create(pwndPropSheetHolder,
                        WS_CHILD | WS_VISIBLE, 0))
        {
                delete m_pFormViewPropSheet;
                m_pFormViewPropSheet = NULL;
                return;
        }
// fit the property sheet into the place holder window, and show it

        CRect rectPropSheet;
        pwndPropSheetHolder-> GetWindowRect(rectPropSheet);
        m_pFormViewPropSheet ->SetWindowPos(NULL,0, 0, rectPropSheet.Width(),
                rectPropSheet.Height(), SWP_NOZORDER | SWP_NOACTIVATE);
}
```

This *OnInitialUpdate()* member function is called by the framework after the view is first attached to the document, but before the view is initially displayed. The *OnInitialUpdate()* function of the *CLectureFrmView* class first calls the OnInitialUpdate() of its base class to carry out the default initialization job. Then calls the *ResizeParentToFit()* to fit the form view to the client area of its attached frame.

Next, the *OnInitialUpdate()* function gets the picture holder object by its ID which is put into the dialog resource of the form view. Following that, it calls the *Create()* function of the property sheet class to create the property sheet. Upon getting the dimensions of the bounding rectangle of the picture holder object, the *SetWindowPos()* is called to set the position and size of the property sheet.

After the setup of the above classes for the Lecture Window, we need to set the position of the lecture window, as the MDI lecture windows show in cascade in the main frame.

The code needed to change the child window position are added to the *CLectureChildFrm::PreCreateWindow(CREATESTRUCT & cs)* function:

```
cs.x=0;
cs.y=0;
```

The *PreCreateWindow* function is called by the framework before the creation of the Windows window attached to the object. *CREATESTRUCT* is a structure that defines the initialization parameters passed to the window procedure of an application. The codes above set lecture windows to the left top of the client area of the main frame when they are created.

The result of the Lecture Window is shown in Figure 5-2:

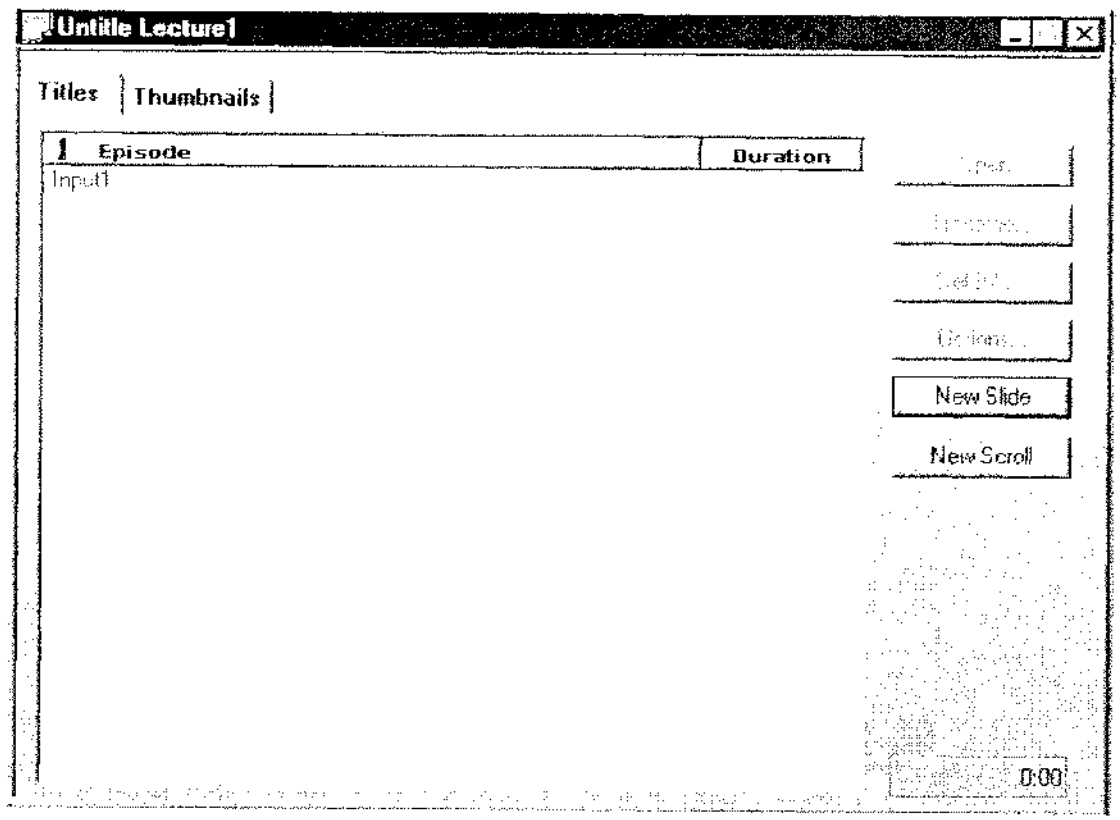


Figure 5-2. The Lecture Window

5.4.2 The Slide Window

The slide window in the application is actually the working area, used to displays the data of a slide. Therefore, for each slide window there is a document for storing the data. Due to the reason, the document/view architecture in MFC is used to dynamically create MDI slide window and attach each slide window to a document.

The data of a slide is different from the data in the document of the lecture window which is just an array of string for the names of a lecture slides. A new type of document class is needed to hold the slide data. For the sake of storing and retrieving a lecture data, each lecture document object has a pointer list that contains the pointers of all the slide document objects.

The creation of slide windows uses the document/view architecture in MFC. The following classes are needed for the application:

- *CSlideDoc*—the slide document class derived from *CDocument*.
- *CSlideChildFrm*—the MDI slide child frame derived from *CMDIChildWnd*.
- *CSlideFrmView* - the slide window view which covers the client area of the slide child frame. As the scrolling ability required for the view, it is derived from *CScrollView*.

Because the application needs to support scrollers and the size of slides varies, the dynamic size of document need to be supported in the application.

To apply the document/view architecture for the slide window creation, the following codes in the gray background need to put in the *CAudioGraphApp ::InitInstance()* function of the application class:

```
CAudioGraphApp::InitInstance()
{       ...
        AddDocTemplate(pAudioDocTemplate);

      CMultiDocTemplate* pSlideDocTemplate;
      pSlideDocTemplate = new CMultiDocTemplate(
                        IDR_SLIDE,

                        RUNTIME_CLASS(CSlideDoc),
                        RUNTIME_CLASS(CSlideChildFrm),
                        RUNTIME_CLASS(CSlideFrmView));

        ...
        if (!ProcessShellCommand(cmdInfo))
        return FALSE;

      AddDocTemplate(pEditDocTemplate);

        ...
}
```

The added code construct a new document template first, then the document template is added in the template list of the application object. The reason for putting *AddDocTemplate* after *ProcessShellCommand* is that the application needs to create a new lecture window at the start by processing the default *File|New* command. Otherwise, the application will search the template list and show a dialog box to let users choose one kind of document between the two types of documents.

Next we will show how to the classes implemented.

The CSlideDoc class

To support the dynamic size of the document, the following members are declared in the *CSlideDoc* class:

```
protected:
   CSize m_DocSize;       // Document size
public:
  CSize GetDocSize()      // Retrieve the document size
      { return m_DocSize; }
```

During the object operations such as inserting a image and drawing, the document size(m_DocSize) must be set if the object size is larger than the document size.

The CSlideChildFrm class

Derived from *CMDIChildWnd* class, the slide window has a minimize box, maximize box, and thick border in order to resize the slide window by dragging on the border. However, the default position and size of the slide window need to be adjusted. The following codes are inserted in the *PreCreateWindow(CREATESTRUCT& cs)* function of the *CSlideChildFrm* class:

```
cs.x=0;  // x coordinate
cs.y=0; //y coordinate
cs.cy = (int)((::GetSystemMetrics(SM_CYSCREEN) )*3.8/ 5); //height
cs.cx = (int)((::GetSystemMetrics(SM_CXSCREEN))*3.8 / 5); //width
```

The above codes set the slide windows position on the left top of the client area of the main window, and the size of slide windows is set to a certain rate of the user's computer screen size in spite of different size of a screen that a computer has.

The CSlideFrmView class

To support the scrolling ability with the dynamical document size, the following code is added in the *OnInitialUpdate()* member function:

```
CSlideDoc* pDoc;
pDoc = GetDocument();
```

```
ASSERT( pDoc!= NULL);
SetScrollSizes(MM_TEXT,pDoc->GetDocSize() );
```

After the view is first attached to the document before the view is initially displayed, *OnInitialUpdate()* is called. *OnInitialUpdate()* first get the document object. Then it calls *SetScrollSizes* to adjust scrolling characteristics. The document size as a parameter is passed to the *SetScrollSizes* function, which results in the recalculation of the size of the view port. If the view size is less than the document size, a horizontal or a vertical scrollbar will show up.

The slide window has four buttons which are positioned against a horizontal scrollbar. The *CSlideFrmView* object is responsible for the creation and management of the buttons and the horizontal scrollbar.

A scroll view in Windows has a default horizontal scrollbar whose fixed length is the same as the width of the view, and a default vertical scrollbar that has the same height as the view does. A default scrollbar is called *windows scrollbar*. Depending on the document size, a scroll view manages whether its *windows bars* show or hide. In the case of the slide window, the horizontal scrollbar is not as long as the view, so the *CSlideFrmView* class needs to manage the position and length of its horizontal scrollbar. But MFC fails to provide the interface functions to access the *windows scrollbars* of a scroll view object. The scroll view class (*CScrollView*) of MFC has a virtual function(*CScrollView::GetScrollBarCtrl*) to deal with scrollbars, which simply returns a *NULL pointer* except in the case of splitter window(such as Visual C++ bitmap editor that divide a window logically in two resizable areas).

To solve the position and size problem of the horizontal scrollbar in the scroll view for the slide window, we will look at the process of the creation of the *CSlideFrmView* class object.

After a *CSlideFrmView* class object is created, but before the object is initially displayed, *CSlideFrmView:: OnInitialUpdate()* is called. This function in turn, makes a call to *CScrollView:: SetScrollSizes()*, which will pass over one of its parameters (the related document size). If the document size is larger than the *CSlideFrmView* view object, it results in the adjustment of the view size to leave spaces for the view's *windows scrollbars*. During the process of *SetScrollSizes()*, *CSlideFrmView::GetScrollBarCtrl* is called the first time to help decide whether the view size needs to be adjusted or not. If the return value of *CSlideFrmView::GetScrollBarCtrl* equals to *NULL*, the scroll view needs to be resized to contain scrollbars. *CSlideFrmView::GetScrollBarCtrl* is called a second time to decide whether to set up *windows scrollbar* or not. If the function returns a pointer of a scrollbar object, the scrollbar will be set up for the scrolling ability, according to the document size. Otherwise, the *windows scrollbars* will be setup for the scroll view, which are not accessible.

The solution for the horizontal scrollbar in the slide window then is to override the *CSlideFrmView::GetScrollBarCtrl* so that the function will return a pointer to a control scrollbar object (*CScrollBar class*), if it is a horizontal scrollbar. When the *GetScrollBarCtrl* function is called by the framework, the results are that the horizontal size of the view object remains unchanged, and the control scrollbar is set up by the framework.

To implement the specified view object of the slide window, the declaration in the *CSlideFrmView* is made as follows:

```
class CSlideFrmView : public CScrollView
{...
    protected:
        CBitmapButton m_Bn_Grid;  //image grid button
        CButton m_Bn_Tools;       //tool button for managing tool window
        CButton m_Bn_EditConsole; // button for managing edit console window
        CButton m_Bn_Attributes;  //button for managing tool window
        CScrollBar* m_pHScrollBar; //the horizontal control scrollbar
...}
```

The grid button is an image button, and the other buttons are standard pushbuttons. When a *CSlideFrmView* object is created, the buttons and the scrollbar need to be created. The code for creating the controls is put in the message handler *OnCreate*, which is called after the scroll view is created but before it becomes visible. The code for *OnCreate* is listed below:

```
/***********Listing 5-5***********/

int CSlideFrmView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{       if (CScrollView::OnCreate(lpCreateStruct) == -1)
                        return -1;
//calculation the position and size for the controls
        ...
        GetParent()->GetClientRect(rect);
        .../create the scrollbar
        if ((m_pHScrollBar->Create(SBS_HORZ|WS_VISIBLE|WS_CHILD|BS_FLAT
                                ,rect1,this, IDC_HSCROLLBAR) ==-1)
                return -1
//create the tool, edit console, and attribute push buttons
        if(m_Bn_Tools.Create("Tools",WS_VISIBLE|WS_CHILD|BS_PUSHBUTTON
                |BS_FLAT, rect2, this,IDC_BN_TOOLS)    == -1)
                 return -1;
        ... // omit the codes for the Edit condole and attribute buttons
//create the grid image button
        if (m_Bn_Gride.Create(NULL,BS_OWNERDRAW|WS_VISIBLE
                        |WS_CHILD, rect3 ,this,IDC_BN_GRIDE))
                return -1;
        //load the resource for the grid button
        m_Bn_Gride.LoadBitmaps(IDB_GRIDE_UP, IDB_GRIDE_DOWN,0,0) ;
        m_Bn_Gride.SizeToContent(); //adjust the image button size

        return 0;
}
```

After completing the default initalization works, the *OnCreate* function uses the parent client area size to calculate the position for the controls. It then in turn calls the *Create* member function for each controls needed to be created. At last the function calls *LoadBitmaps* and *SizeToContent* to load the specified bitmap resource for the grid button and adjust the button size.

The *Create* method of *CScrollBar* have four parametres to specified the scrollbar's styles, position and size, parent, and ID. The Create function of the buttons has one more additional string parameter used for the caption of the button.

The controls have some common styles: *WS_CHILD*(indicating the object is a child window). *WS_VISIBLE*(indicating the object is visible). Other styles used in the *OnCreate* function are:

- SBS_HORZ—indicate the scrollbar is horizontal scrollbar.
- BS_PUSHBUTTON—the button is pushbutton.
- BS_FLAT—indicate a flat button.
- BS_OWNERDRAW—for bitmap buttons.

The LoadBitmaps method has four parameters, which are bitmap resource IDs in turn loaded to the bitmap button when it is in the normal or "up" state, selected or "down" state, focused state, or disabled state.

To deal with the scrollbar, we override the *GetScrollBarCtrl* function:

```
/************Listing 5-6 ************/

CScrollBar* CEdit_View::GetScrollBarCtrl(int nBar) const
{
        if (nBar ==SB_HORZ)
                return m_pHScrollBar;
        else
                return CScrollView::GetScrollBarCtrl(nBar);
        }
```

The Parameters *nBar* specifies the type of scroll bar. The parameter can take one of the following values:

- *SB_HORZ*: Retrieves the position of the horizontal scroll bar.
- *SB_VERT* : Retrieves the position of the vertical scroll bar.

If the parameter of the *GetScrollBarCtrl* indicates a horizontal scroll bar, the function returns a pointer to the control scroll bar *(m_pHScrollBar)*. Otherwise it calls the default function of the base class(*CScrollView),* which in this case returns *NULL.*

There are still some concerns about the controls in the scroll view. As the scroll view is resized or its scroll bars clicked, the scroll view is scrolled with the result that the controls moves on the scroll view. There are the three message handlers for the scroll view to deal with this situation:

- CScrollviewView::OnSize—The framework calls this member function after the window's size has changed.
- CScrollviewView::OnVScroll— The framework calls this member function when the user clicks the window's vertical scroll bar.
- CScrollviewView::OnHScroll— The framework calls this member function when the user clicks a window's horizontal scroll bar.

The above three inherited functions need to be overridden. As the three functions do quite similar jobs in dealing with the position of the controls in the scroll view, here just lists the code for *OnVScroll:*

/**********Listing 5-7 ************/**

```
void CScrollviewView::OnVScroll(UINT nSBCode, UINT nPos, CScrollBar*
                                                    pScrollBar)
{       ...//omit for the calculation the positions
        InvalidateRect(rect,true);  //invalidate the view area
        CScrollView::OnVScroll(nSBCode, nPos, pScrollBar);
// change the position of the controls
        m_pHScrollBar->MoveWindow(rect1, true);

        ...//omit for the buttons   }
```

OnVScroll calls *InvalidateRect* first to invalidate the client area of the scroll view for repainting, and then moving the controls position by calling *MoveWindow.*

The result slide window is shown below:

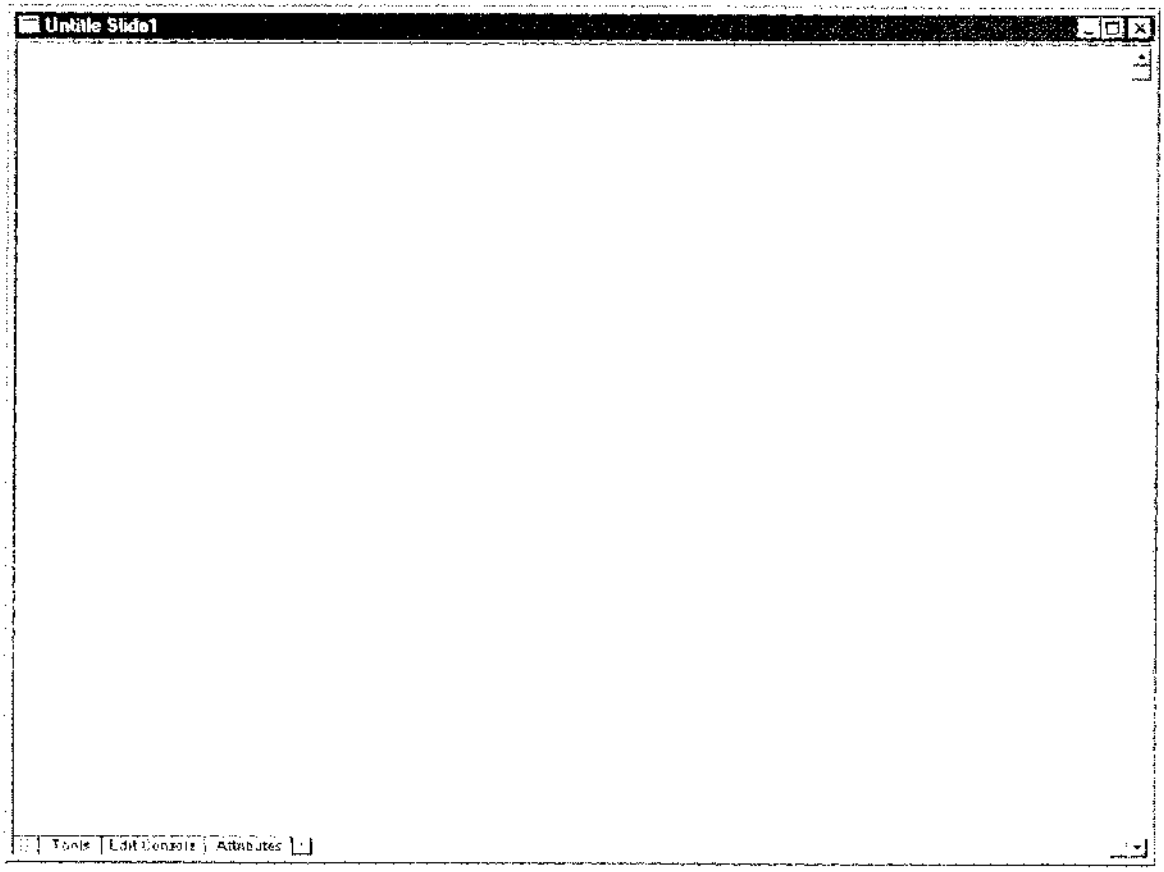


Figure 5-3. The Slide Window

The horizontal scroll bar in the slide window is never hidden, even if the document size is smaller than the view port size which just results that the horizontal scroll bar is disabled. The bottom right of the slide window doesn't have a resize area. However, it is not a problem for resizing the slide window, because the border of the slide window can be dragged to achieve this goal.

5.4.3 The Attributes Window

The attribute window is used as a tool window for the application and it has different controls such as bitmap buttons, slide control and so on. The controls need to pass messages as commands to the command hierarchy. The *CDialogBar* class(dialog bar) has been chosen for the attribute window object. However, the *CDialogBar* object can't

handle bitmap buttons, so a class *CInitDialogBar* needs to derived from *CDialogBar* with the additional functionality to handle bitmap button. The class (*CAttributeDialogBar*) for the attribute window object can then be derived from the *CInitDialogBar* class.

The bitmap button initialization happens after the dialog bar creation. The interface provided by MFC is a message handler *CDialogBar::OnCreate()* which happens before the *CDialogBar::Create()* function returns. To handle bitmap buttons, the solution is to override the *CDialogBar::Create()* function in order to have some time to initialize the bitmap buttons before *CDialogBar::Create()* returns. The code for the *CInitDialogBar*:: Create() function is listed below:

```
BOOL CInitDialogBar::Create(CWnd * pParentWnd, UINT nIDTemplate,
        UINT nStyle, UINT nID)
{       if(!CDialogBar::Create(pParentWnd,
                MAKEINTRESOURCE(nIDTemplate), nStyle, nID))
                return FALSE;
        if(!OnInitDialogBar())      return FALSE    ;
        return TRUE;
}
BOOL CInitDialogBar::OnInitDialogBar()
{       UpdateData(FALSE);
        return TRUE; }
```

The *CInitDialogBar::Create()* calls its base class *Create* function to carry out the default creation of the dialog bar , then calls *CInitDialogBar:: OnInitDialogBar(),* which in turn calls UpdateData() to initialize data in the dialog bar, or to retrieve and validate dialog data.

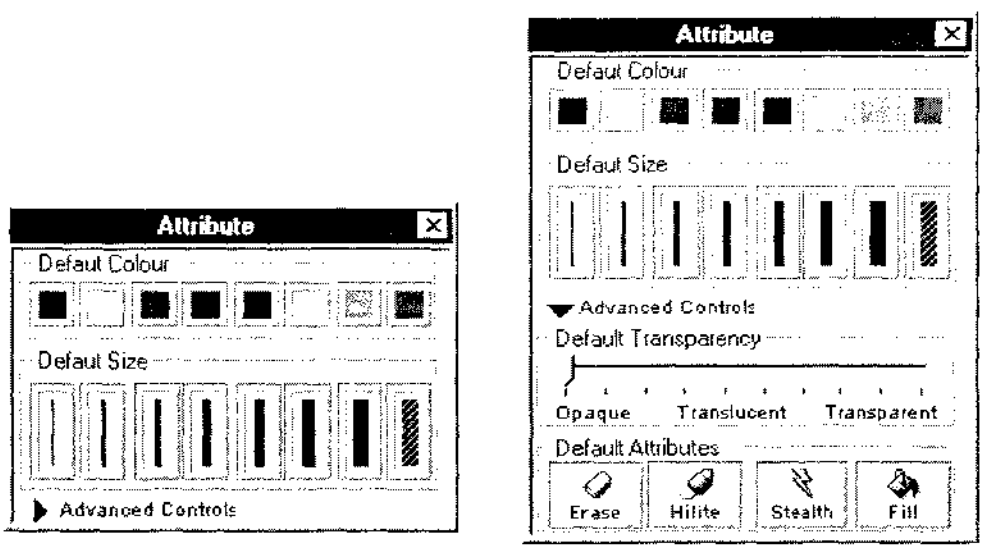
In the *CAttributeDialogBar* class derived from *CInitDialogBar*, the *CInitDialogBar:: OnInitDialogBar()* function is overridden in order to initialize the bitmap buttons:

```
BOOL CAttributeDialogBar::OnInitDialogBar()
{       CInitDialogBar::OnInitDialogBar();
```

```
    // slide control set up
    CSliderCtrl* pSlider = (CSliderCtrl* ) GetDlgItem(IDC_SLIDER1);
    pSlider->SetRange(0,9);
    //buttons set up
    VERIFY(m_BnErase.SubclassDlgItem(IDC_ERASE, this));
    m_BnErase.LoadBitmaps(IDB_ERASE_UP, IDB_ERASE_DOWN ,0,0) ;
    m_BnErase.SizeToContent();
    ...
  //omit the control font set up
}
```

When the attribute tool window object is created by its *Create() function*, the default creation process for the dialog bar is finished first, then *OnInitDialogBar()* is called to give time to initialize the controls(especially for bitmap buttons).

The result Attribute Window is shown below:



Fogure 5-4. The Attribute Window

The implementation of the Tool window is the same way as the attribute window does. Figure 5-5 shows the result interface of the Tools Window.

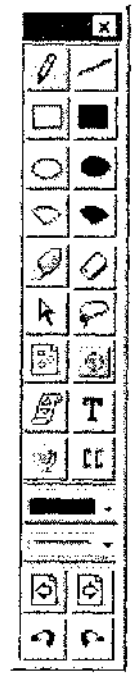


Figure 5-5. The Tool Window

5.4.4 The Edit Console Window

The edit console has a toolbar, and a scroll view where icons are inserted when an annotation happens on the slide window. The classes for the edit console window:

- *CEditConsoleFrm* —the window frame object derived from *CMiniFrameWnd*. A *CMiniFrameWnd* object represents a half-height frame window typically seen around floating toolbars.
- *CEditConsoleView* -the scroll view of the window, derived from the *CScrollView*.

To achieve the dynamic size of the scroll view, the *CEditConsoleView* class has a member for dynamical size: *CSize m_ViewSize*. In the constructor, the dummy view size is set:

```
SetScrollSizes(MM_TEXT, CSize(1,0));
```

A function is declared in the class in order to set the view size under the condition that the buttons inserted in the view are over the view port:

```
CEditConsoleView::SetViewSize(CSize size) { m_ViewSize=size;}
```

The *CEditConsoleView* class has a member of a pointer to bitmap button whose ID number is dynamic. To hold the a collection of bitmap buttons in the edit console window, we need to put the following declaration in the *CSlideDoc* class:

```
CTypedPtrList< CbitmapButton, CbitmapButton*>m_pBitmapBnList
```

This is a pointer list for bitmap button. If an annotation is happen in the slide window with the left button of the mouse up, a corresponding bitmap button is created in the edit console and inserted in the list of the slide document for retrieving in the future.

In the *CEditConsoleFrm* class, two members for the toolbar and the *CeditConsoleView* is declared as follows:

```
public:
        CEditConsoleView m_EView;
        CToolBar m_EToolBar;
```

The creation codes of the toolbar and the view are put in the OnCreate() function, which is called after the creation of the edit console frame but before it becomes visible. The code is listed below:

```
/*********************Listing 5-8 ***************************/

int CEditConsoleFrm::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CMiniFrameWnd::OnCreate(lpCreateStruct) == -1) return -1;
//get the position and size of the view
        CRect rect;
        GetClientRect(rect);
        rect.bottom -=30;
//create the view
        if (!m_Eview.Create(NULL,"EditConsoView",WS_CHILD|WS_VISIBLE,
                                rect,this,IDC_EC_VIEW,NULL))
        {       TRACE0("Failed to create view\n");
                return -1;      // fail to create
        }       //create the toolbar
        if (!m_EToolBar.Create(this, WS_CHILD | WS_VISIBLE | CBRS_BOTTOM )
                        ||!m_Consol_menu.LoadToolBar(IDR_EDITCONSOLE_BAR))
        {       TRACE0("Failed to create debugbar\n");
                return -1;      // fail to create
        }
        return 0;
}
```

The view object and the tool bar are specified as child windows and are visible at the start. The tool bar aligns at the bottom of the frame. The *CToolBar::Load* function loads the specified tool bar resource to the bar.

To handle the clicking of the icon button inserted in the Edit Console, we need to code the handler for the clicking of the left button of the mouse. The code is listed below:

```
/*********************Listing 5-9 ***************************/

void CEditConsoleView::OnLButtonDown(UINT nFlags, CPoint point)
{
   CSlideDoc* pDoc;
...//omit the code to get the document

POSITION aPos = pDoc->GetBitmapBnListHeadPosition();
CBitmapButton*  pBitmapBn =NULL;

//iterate the bitmap button list in the document to find which button is clicked
While(aPos){

pBitmapBn = pdoc->GetNext(aPos);
RECT *rect;
rect =pBitmapBn->GetWindowRect;
if (rect-> PtInRect( point )){                  //by point test to find the bitmap button
        pBitmapBn->SetState( true);  //load the "down" state bitmap
        // get the position for visual component for playback
        m_VisualComponentPosition = aPos;
}
else
        pBitmapBn->SetState( false);  //set other buttons in "up" state
}
}
```

When clicking in the Edit Console view, the message handler first is to get the document object. Then iterate the bitmap button list stored in the document to find the right bitmap button which is clicked. At the same time, set all the bitmap buttons states, as well as the member m_VisualComponentPosition of the POSITION type, which is needed to passed to the CSlideFrmView in the case of drawing the visual components for playback.

The result interface of he edit console window is shown below:

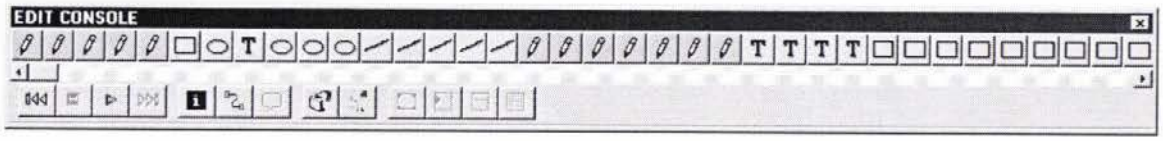
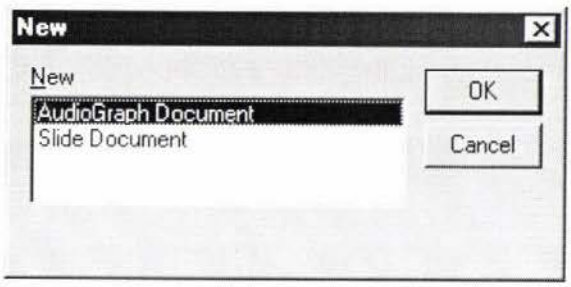


Figure 5-6. The Edit Console

5.4.5 Manage the two document type windows



Because there are two types of documents in the application, the application has two document templates in its template list. As discussed in chapter 4 about the document/view architecture, a default dialog box appears when users click *File|New* menu item. After selecting a document type in the dialog, a lecture or slide window will appear. The default dialog is shown below:

As the consistence with the application in Macintosh, the default dialog needs to be changed as shown in Figure 5-7:

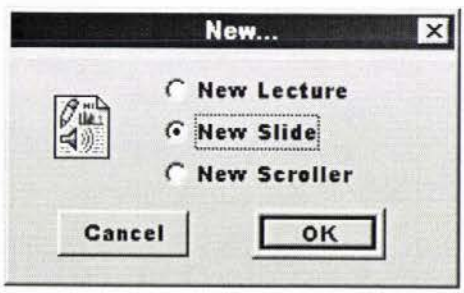


Figure 5-7. the document selection dialog

The solution is constructing the wanted dialog. After checking the *OK* button is clicked, search the document template in the application object to find the proper document template. Then call *OpenDocumentFile* function of the application class to continue the rest of the process in the document/view architecture.

The method also is used to create new slide window by clicking the New Slide button of the lecture window and opening a slide window by double clicking one existing slide

name in the list of the Lecture Window. Here just look at the codes by clicking *File|New* menu item.

Although the message can be managed by the main frame object. But the CWinApp object has the list of the document templates and it has functions to manage the list, We add a message handler *OnFileNew* in the *CAudioGraphApp* class. The code is shown in Listing 5-10:

```
/**********************Listing 5-10************************/

void CAudioGraphApp::OnFileNew()
{
//hide the Tools, Attribute, Edit Console windows if there are created already
CMainFrame* m_pMainWnd = GetMainWnd(); //get the main farme
if (m_pMainWnd -> m_pAttributeWnd != NULL)
        m_pMainWnd -> m_pAttributeWnd->ShowWindow(SW_HIDE);
 ...//omit the codes to hide the tool and edit console window

CFileNewDlg  aDlg;  //local construct the selection dialog
if( aDlg.DoModal() == IDOK )  //OK button is clicked or not
{
        // Searches template list for a document type
        POSITION curTemplatePos = GetFirstDocTemplatePosition();

         while(curTemplatePos != NULL)
        {
            CDocTemplate* curTemplate = GetNextDocTemplate(curTemplatePos);
            CString str;
            curTemplate->GetDocString(str, CDocTemplate::docName);

        //find the lecture dodument
            if(str == T("Lecture Document") && aDlg.m_nDocType == LECTURE)
            {
                curTemplate->OpenDocumentFile(NULL);//continue the process in doc/view architecture
                return;
            }

        //find a  slide document
             if(str == T("Slide Document") && aDlg.m_nDocType == SLIDE)
            {
        //check three tool window created or not. If not, create them, otherwise show them
                //if edit console window is not create, create it
                ...//omit size calculation
                if (m_pMainWnd -> m_pEditConsoleWnd == NULL)
                {
                    m_pMainWnd -> m_pEditConsoleWnd = new CEditConsoleFrm;
                    if (!(m_pMainWnd -> m_pEditConsoleWnd) ->
                        Create(NULL,"EDIT CONSOLE", WS_POPUPWINDOW|WS_CAPTION |
                                WS_VISIBLE,   aRect,m_pMainWnd))
                        return -1;
                }else
                   m_pMainWnd -> m_pEditConsoleWnd ->ShowWindow(SW_SHOW);

                ...//omit the other two window

                ...//omit the code to find the frame ->view->property sheet->the list page
                ...//insert string in the list of the lecture window
                 }
             }
}
```

In the code of listing 5-9, the first task is to hide the three tool windows as the *File|New* menu item is clicked, and then gets the position of the first document template in the application. Use the *POSITION* value returned in the call *(GetFirstDocTemplatePosition( ))* to the *GetNextDocTemplate* function to get the first CDocTemplate object, to start the search of the document list. If the selection in the dialog box is new lecture, then create a new lecture window. Otherwise create the new slide window. The three tool windows are created in the OnFileNew() when the first slide window open.

After the above discussion of the interface implementation, next we will go to the implementation of some functionality (drawing) in the application.

5.4.6 The Implementation of some functionality

Visual components include line, curve(for pen tool) rectangle, oval and circle. They have color and pen width attributes. For rectangle, circle and oval, they also have a brush attribute in order to fill them. This section will discuss implementing drawing line, curve and rectangle with different colors and pen widths, as well as inserting text in the Slide Window.

The line, curve and rectangle classes and their base class *(CVisualComponent)* can be ported from the Mactiontosh version. Some modification needs to be made, especially in order to support serialization, the component classes must derived from *CObject*.

The declaration of the *CVisualComponent* class is shown below:

```
/**********************Listing 5-11************************/

class CVisualComponent: public CObject
{
DECLARE_SERIAL(CVisualComponent)

protected:
  COLORREF m_Color;              // Color
  CRect m_EnclosingRect;         // Rectangle enclosing an component
  int m_PenWidth;                // Pen width
 bool
public:
  virtual ~ CVisualComponent (){}              // Virtual destructor

  // Virtual draw operation
  virtual void Draw(CDC* pDC){};
    CRect GetBoundRect();        // Get the bounding rectangle for an element

  virtual void Serialize(CArchive& ar); // Serialize

protected:
  CVisualComponent (){}                    // Default constructor
};
```

The *CVisualComponent* class has color, pen width, and rectangle attributes. The rectangle attribute defines a rectangle area that contains a component. It has a virtual *draw()* function that make sure the correct override *draw()* in the derived class is called in question. It also has a virtual destructor, which ensures the derived class object is destroyed properly. The parameter *CDC* for the *draw()* function is a device context class that provides things to be needed by way of graphical output. A DECLARE_SERIAL macro is added in the class for supporting serialization.

The *CLineComponent* defines as:

```
/**********************Listing 5-12************************/

class CLineComponent: public CVisualComponent
{
DECLARE_SERIAL(CLineComponent)

public:
  virtual void Draw(CDC* pDC);/ // Function to display a line

  // Constructor for a line object
  CLine(CPoint Start, CPoint End, COLORREF aColor, int PenWidth);

  virtual void Serialize(CArchive& ar); // Serialize function for CLine

protected:
  CPoint m_StartPoint;        // Start point of line
  CPoint m_EndPoint;          // End point of line

  CLineComponent (){}
};
```

The data members(*m_StartPoint* and *m_EndPoint*) define a line object. It has a constructor with two points, color and pen width parameters that are used to pass value from other objects when a line object is constructed.

The declaration for other derived classes from *CVisualcomponent* are similar to the *CLineComponent* except they have no the two point members. Additionally, CCurveComponent class has the following members:

```
CList<CPoint, CPoint&> m_PointList;
void AddSegment(CPoint& aPoint);
```

As a curve consists of small segments of line, the pointer list is used to store all the points for the curve.

The *CTextComponent* has additional members for a text string and start position:

```
protected:
  CPoint m_StartPoint;          // position of a text component
  CString m_String;             // Text to be displayed
```

The syntax for making a component in the slide window view is:

1. Record the start point when the mouse left button clicked; if it is a text component, display a dialog that lets the user enter text, and create a bitmap button in the edit console window. Then add a new string to the component list in document.
2. Draw the component as the mouse moves.
3. When the mouse moves, record the second point. If it is a curve component, add the point to the point list of the curve object, and draw the component.
4. When the mouse left button is up, add the component to the component list of the document, and create a corresponding bitmap button in the edit console window.
5. When resizing the slide window, Windows invalidates the whole client area so that the drawing all disappear. At this time, Windows sends a WS_PAINT message that will lead to a call of the *OnDraw* function of the view class. Therefore we need to draw the components stored in the component list of the document by the *OnDraw* function.

Let's look at the implementation by starting with the user chooses a color. The handler for the single-clicking message is in the CSlideDoc class. The code is listed below:

```
CSlideDoc::OnRed()
{       //get a pointer to the red color button, //then load the down-state bitmap to the button
        CBitmapButton* RedBn = GetDialogItem(IDC_RED_COLOR);
        RedBn->SetState(true);
...//omit the code for set other buttons states
m_Color=RED;  //store the specified color in the member variable
                        //RED is defined macro( COLORREF RED = RGB(255,0,0)) }
```

When the user clicks the red color button in the Attribute Window, the handler *OnRed()* set up the bitmap for the bitmap buttons. Then store the red color to its member.

When the left button of the mouse is down within the view area of the slide window, a message is sent to the view object. The code for the message handler is listed below:

```
/**********************Listing 5-12************************/

void CSlideFrmView::OnLButtonDown(UINT nFlags, CPoint point)
{
   CClientDC aDC(this);              // Create a device context for drawing in client ara
   OnPrepareDC(&aDC);                 // Get origin adjusted
   aDC.DPtoLP(&point);                // convert point to Logical

  CSlideDoc* pDoc = GetDocument();  // Get a document pointer
   if(pDoc->GetElementType() == TEXT)
   {
     CTextDialog aDlg;
     if(aDlg.DoModal() == IDOK)
     {
         ...//omit codes for creating a button in the edit console
         ...//omit codes for the calculatin the botton right point
         CTextComponent* pTextElement = new CTextComponent(point, BottomRt,
                        aDlg.m_TextString, pDoc->GetElementColor());

       // Add the element to the document
       pDoc->AddElement(pTextElement);

       // Get all views updated
       pDoc->UpdateAllViews(0,0,pTextElement);
     }
     return;
   }
   m_FirstPoint = point;            // Record the cursor position
   SetCapture();                  // Capture subsequent mouse messages

}
```

In the function, the first task is to create a device context object, then convert the cursor position from device unit to logic unit (in logic unit the top left of the client area is (0,0)). If it is a text component, then construct a dialog box. After creating a string, add it to the object list of the document, and update all views. If not, simply store the cursor position. The *SetCapture* function is used to cause all subsequent mouse input to be sent to the object regardless of the position of the cursor.

The codes for the mouse-move handler in the CSlideFrmView class are:

```
/**********************Listing 5-13************************/

void CSlideFrmView::OnMouseMove(UINT nFlags, CPoint point)
{ ...
  if((nFlags & MK_LBUTTON) && (this == GetCapture()))
  {
    ...
    m_SecondPoint = point;     // Save the current cursor position

    if(m_pTempComponent)
    {
      if(CURVE == GetDocument()->GetComponentType())  // Is it a curve?
      { // drawing a curve
        // so add a segment to the existing curve
        ((CCurve*)m_pTempElement)->AddSegment(m_SecondPoint);
        m_pTempElement->Draw(&aDC); // draw it
        return;
      }
        ...
    }
    // Create a temporary component of the type and color that
    // is recorded in the document object, and draw it
    m_pTempComponent = CreateComponent(); // Create a new element
    m_pTempComponent->Draw(&aDC);       // Draw the element
  }
}
```

The function stores the cursor position to a member *m_SecondPoint*. Then if it is drawing a curve, add the point to the point list of the curve object and draw it. If not, create a temporary component of the type and colour that is recorded in the document object, and draw it. The *CreateComponent()* function calls the constructor for a component type, such as:

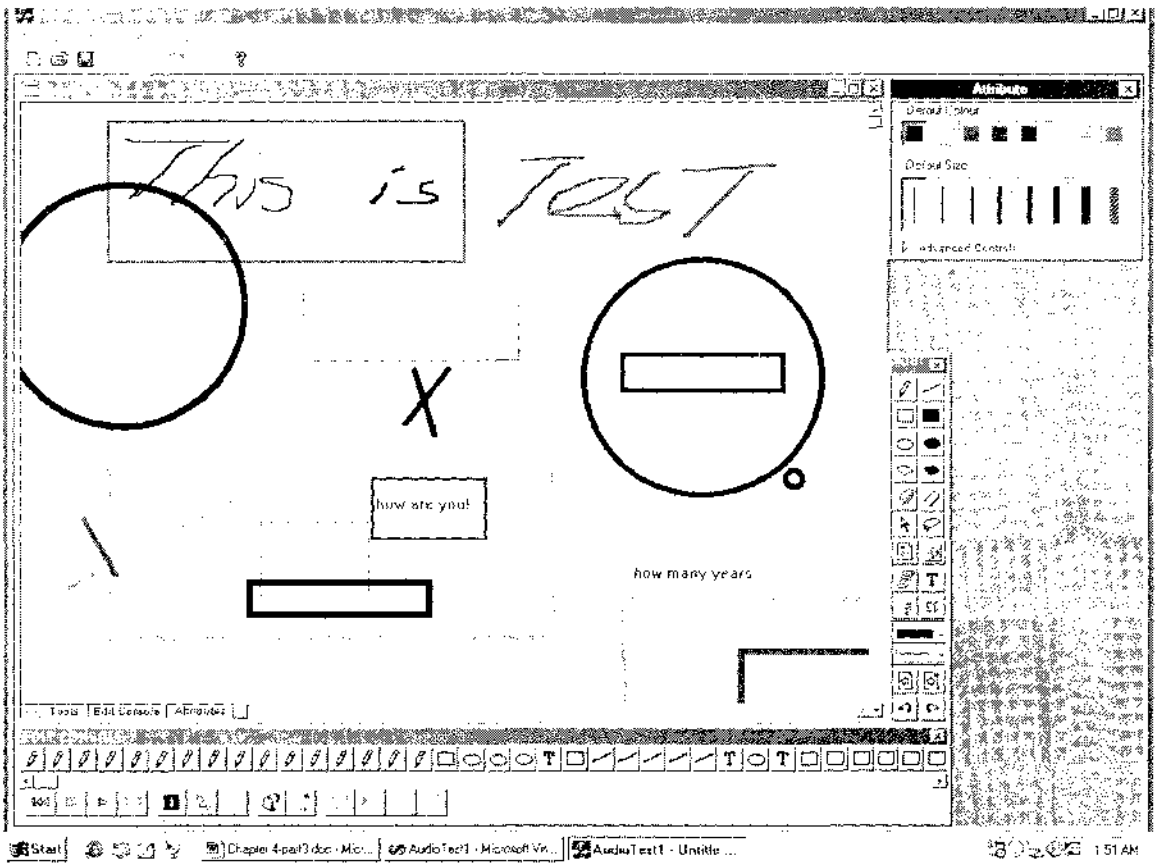
```
case RECTANGLE:
   return new CRectangle(m_FirstPoint, m_SecondPoint,
                                        pDoc->m_Color, pDoc->m_PenWidth);
```

pDoc is a pointer to the CSlideDoc object.

In the handler *CSlideFrmView::OnLButtonUp* for the mouse-up message. It creates a bitmap button in the edit console. According to the position of the button, set the size of the view object of the edit console. The document size is set by the point of the cursor. And then add the object to the object document by the code:

```
GetDocument() >AddComponent(m_pTempComponent);
```

Here is the interface after drawing some components in the slide window:



Summery

In this chapter, we discussed the implementation of some specific windows and showed the solutions to some problems encountered during the implementation of the application. we also demonstrated how to port and modify some code from the Macintosh version application by the implementation of some visual component.

Chapter 6 Conclusions

Like many software projects, code if often implemented without proper specification or documentation. The Macintosh AudioGraph was in this category, consequently I have gained a lot of knowledge about the method of reverse engineering and also about application frameworks on two platforms. In particular I have gained knowledge in the following fields:

- The general design patterns of an application framework for event driven programming. These include the application, message system, persistence, and class hierarchy etc.
- The PowerPlant application framework and how it is implemented, especially the visual hierarchy, command hierarchy and broadcaster-listener message system of PowerPlant.
- The MFC application framework and its internals, especially its mechanisms such as document/view architecture, message mapping system, serialization, class hierarchy and the process of a window object creation, and how functions are called in sequences.

6.1 Achievement

The interface of a PC version AudioGraph Recorder is constructed using MFC in Visual C++ 6.0 and tested in current available Windows OS. The interface achieved is, as specified by my supervisor, completely consistent with that of the Macintosh version. Additionally, some functionality has also been implemented, such as drawing different shapes with different colours and pen widths, and inserting text. The design and implementation of the interface takes full advantage of the current support features of MFC, and extends some of them where required. The solution not only considers the interface part, but also focuses on the interface compatibility with any future

development to this project, for example the implementation of the remaining functionality.

6.2 The implementation difficulties

As this project is about porting an existing project from the Macintosh to PC, reverse engineering techniques have been applied throughout the development. The reason for this was that the implementation documentation for the Macintosh AudioGraph Recorder was not available, hence all the information required normally for designing good software had to be extracted from the source code that is very large.

PowerPlant is completely different from MFC. Especially in the inheritance use for their framework implementation, PowerPlant adopts the OO multiple inheritance technique, whereas MFC uses single inheritance instead; and in the implementation of message system, PowerPlant uses its broadcaster-listener system, in which a broadcast can send messages to multiple listeners. The messages are handle by the listener itself so that a broadcaster does not need to know the listener in advance; in MFC, a message handler is a member function of a class. This has lead to considerable difficulty in remodelling the design from the Macintosh to PC

The PC interface had to be consistent with the Macintosh one, but not all features of the interface components were supported by MFC. To expand the behaviour of a class in the coherent structured MFC framework, the internal implementation of the class must be mastered first.

Most MFC programmers feel difficulty even before they insert one line of their own code, me also. The classes in MFC have too many functions, which must to be called in sequence. And some of the documentation for MFC is very brief. Some help classes used inside MFC are not documented. So every step of the project implementation needs to be tested to prove the expected result. In this sense the programming became an empirical science.

6.3 Testing

The interface of the PC version has been tested in Windows NT, Windows 95, Windows 98 and Windows 2000, and has been shown to be error-free and consistent with the Macintosh one.

In testing, as the user interacts with the interface, a corresponding window or dialog will be opened. The states of control buttons in the Tool Window and the Attribute Windows change accordingly, and the state of menu, including the main menu in the main frame and the two pop menus in the Tool Window also change accordingly, and corresponding text of a menu item are replaced as required. The interface responds to the user interaction just like the Macintosh one does.

The differences between the interfaces of the Macintosh version and the PC version are that the interface of PC version has a main frame, and the Slide Window has no grid area. Also the vertical scrollbar in the Slide Window hides when the document size is smaller than the view port as well as the horizontal scrollbar in the Edit Console.

Aesthetically at least, I personally think that the Macintosh interface components have a much better look, for example, the title bar has grid line but in the Windows window title bar just simply has colour and the controls have a better 3-D look.

6.4 Future works

For the interface part, some bitmap resources need to be constructed, such as the down-state bitmaps for the grid, redo and undo button and recorder button, and the disable-state bitmaps for the previous page and next page button, redo and undo button and pause button. The implementation of these is trivial.

The future work mainly concern the implementation of the application's functionality. Some suggestions are given:

1. Drawing

For dawing arc, derive a class from the CVisualComponent. Use the *Arc()* member of CDC class (device context) to draw it as other component.

For drawing a filled component, such as a filled rectangle, add another parameter (*bool bFill*) in its constructor. When clicking to choose the filled component, the message handler in document can store a flag to indicate it is for filled component. In the *draw()* function of the class, create solid *CBrush* object if the flag is *true*, otherwise create a hollow brush.

2. Serilization

Currently, the data in each Slide Window can be stored to a *.aep* file. To store a whole lecture data to a file, declare a *CSlideDoc* object pointer list in the *CLectureDoc* class. When creating a new slide window, add a pointer of the related document to the list of the *CLectureDoc* class. When serializing the data in *CLectureDoc* object, also serialize the *CSlideDoc* object in the object pointer list.

In conclusion then the AudioGraph recorder interface has been reversed engineered, the frameworks of both MFC and PowerPlant have been mastered and a significant component of the PC version of the recorder has been implemented, namely the interface components, which are those that do not port from one platform to another.

References:

- **Book**

[A1] Linda M.Harasim, 1990, "online education: perspectives on a new environment", Praeger Publishers, US. ISBN 0-275934-489.

[A2] Dave Collins, 1995, "Designing Object-Oriented User Interfaces", the Benjamin/cummings Publishing Company, Inc. ISBN 0-8053-5350-x.

[A3] Ivor Horton, 1997, "Beginning MFC programming", Wrox Press Ltd of UK. ISBN 1-861000-85-5.

[A4] Chris H. Pappas, 1996, William H. Murray, "Java with Borland C++", Academic Press, Inc. of UK. ISBN 0-12-511960-7.

[A5] David J.Kruglinski, 1997, "Inside Visual C++", Microsoft Press, a Division of Microsoft Corporation, US, ISBN 1-57231-565-2.

[A6] Shepherd, George (George R.) 1996, "MFC Internals", ISBN 0-201-40721-3.

[A7] metrowerks, "Inside PowerPlant for CW11", 1996, Metrowerks Inc.

[A8] George Towner / Apple Computer, Inc. " Discovering QuickTime", 1999, ISBN: 0-12-059640-7.

[A9] William H. Murray, Chris H. Pappas, 1996, " Widows 95 and NT programming with the Microsoft foundation class library", Academic Press, Inc of US. ISBN 0-12-511890-0.

[A10] Viktor Toth, "Visual C++ TM second edition", 1997, Sams Publishing of USA, ISBN: 0-672-31013-9.

[A11] Thomes A.Powell, "HTML: The Complete Reference, Second Edition", Brandon A.Nordin, ISBN: 0-07-211977-2

- **Web Site**

[B1] David P.Diaz, "Technology Training for Educators: The Pedagogical Priority", http://www.sacco.k12.ca.us/pub_webdir/CUE_Newsletters /2Mar_Apr_00_Lead_Article.doc. Visited at 12/00.

[B2] http://www.microsoft.com.Visited at 12/00.

[B3] http://www.adobe.com/products/livemotion/main.html. Visited at 08/00.

[B4] http://www.nzedsoft.co.nz , visited at 12/00.

[B5] Chris Jesshope, 1999, "Quick start documentation". http://www.nzedsoft.co.nz/Documentation/quickstartconten.html Visited at 11/00.

[B6] Chris Jesshope, 1999, "AudioGraph Recorder concepts". http://www.nzedsoft.co.nz/Documentation/conceptscontents.html. Visited at 11/00.

[B7] Diane Gartner, 1999, "A Glimpse at the Benefits of an Object-Oriented Interface" http://pages.cthome.net/iact/IQN/6-1999oct/iqn6-OOI-Benefits.html. Visited at 07/00.

[B8] "An introduction to Macintosh Programming for Windows Programming".
http://developer.apple.com/macos/macoverview/prog01.html.
Visited at 06/00.

[B9] Paul Kuliniewicz, "Introduction to Windows API".
http://www.vbapi.com/articles/intro/part01.html.
Visited at 11/00.

[B10] "An introduction to Macintosh Programming for Windows Programming".
http://developer.apple.com/macos/macoverview/prog07.html#sect7.1.
Visited at 11/00.

[B11] "An introduction to Macintosh Programming for Windows Programming".
http://developer.apple.com/macos/macoverview/prog06.html#sect6.2.
Visited at 12/99.

[B12] "An introduction to Macintosh Programming for Windows Programming".
http://developer.apple.com/techpubs/mac/Files/Files-14.html.
Visited 01/00.

[B13] "NSDocument".
http://devworld.apple.com/techpubs/macosx/Cocoa/Reference/ApplicationKit
/Java/Classes/NSDocument.html
Visited at 11/00.

[B14] "JAVA™ FOUNDATION CLASSES: NOW AND THE FUTURE".
http://java.sun.com/products/jfc/whitepaper.html.
Visited at 01/00

[B15] http://www.ddj.com/articles/1999/9902/9902i/9902i.htm

[B16] "Heuristic Evaluation".
http://www.useit.com/papers/heuristic/
Visited at 10/00.

[B17] "Unified Modelling Language".
http://www.rational.com/uml/index.jsp
Visited 01/01.

[B18] "Rational Software Products Overview".
http://www.rational.com/news/products.jsp
Visited at 09/00.

[B19] "Importers and Exporters".
http://developer.apple.com/samplecode/Sample_Code/QuickTime
/Importers_and_Exporters.htm.
Visited at 11/00.

[B20] "Recording Sounds Directly From Sound Input Devices".
http://developer.apple.com/techpubs/quicktime/qt4beta/INMAC/SOUND
/imsoundinput.19.htm.
Visited at 11/00.

[B21] "Make Your Own Sound Components".
http://developer.apple.com/dev/techsupport/develop/issue20/20olson.html
Visited at 11/00.

[B22] Dr. Dobb. "Comparing WFC and JFC"
http://www.ddj.com/articles/1999/9902/9902i/9902i.htm.
Visited at 10/00.

- **Paper**

[C1] Diaz, D. P. (1999). CD/Web hybrids: Delivering multimedia to the online learner. Journal of Educational Multimedia and Hypermedia. 8(1). 89-98.

[C2] Hilz, Starr Roxanne, 1993, " The virtual classroom: Learning without limits via computer networks", Ablex Publshing Corporation, US.

[C3] Sam Hsu, Oge Marques, M. Khalid Hamza, Bassem Alhalabi, 1999, "How to Design a Virtual Classroom: 10 Easy Steps to Follow", September 1999 issue of T.H.E. Journal .

[C4] Chris Jesshope, Alex Shafarenko and Horia Stusanschi, "Low-bandwidth multimedia tools for Web-based lecture publishing".

[C5] SEGAL, J., 'An evaluation of a teaching package constructed using the AudioGraph, Web-based lecture recorder', ALT J, 1997, **5**, (3), pp.32-42.

[C6] Green Mark, "a survey of three dialogue models", ACM Transaction on Graphics 5,3(July 1986),PP.244-275.

[C7] Charles Richter, 1999, "Designing object-oriented systems with UML", published by Macmillan Technical Publishing.

[C8] Mooney, J., "Strategies for Supporting Application Portability", IEEE Computer, Vol. 23, No. 11, Nov. 1990, pp. 59-70

[C9] Bennett, John. 1977. "User-Oriented Graphics System for Decision Support in Unstructured Tasks." User-Oriented Design of Interactive Graphics Systems. ACM/SIGGRAPH.

[C10] Boehm, Barry W. "A spiral Model of Software Development and Enhancement." Computer (IEEE) 21, 5 (May 1988).

Errata sheet

1. P.14, line 18, "existing application"

2. P.14, line 20, "existing Macintosh version"

3. P.43, section 2.7 "The Preference Dialogue"

4. P.44, Add double quote to "Heuristic evaluation…design process"

5. P142, line 2, "code is often implemented"

6. Referance e[4]: Jesshope,C.,Shafarenko,A.,Slusanschi,H. (1998): "Low-bandwidth multimedia tools for Web-based lecture publishing". IEEE Engineering Science and Educational Journal,9(4),156-162.