

SOFTWARE ARCHITECTURE FOR CROSS
PLATFORM USER INTERFACES

by

BADRIPRASAD KESHAVAMURTHY, B.S.

A THESIS

IN

COMPUTER SCIENCE

Submitted to the Graduate ~~Faculty~~
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

Approved

August, 1996

ACKNOWLEDGEMENTS

I would like to thank Dr. Bagert for agreeing to be my advisor and guiding me through the various stages of my research with utmost patience. Also, I would like to extend my sincere gratitude and thanks to Dr. Lakhani for agreeing to be on my committee. Finally, I would like to thank my family for being a source of inspiration in all my endeavors.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS.....	ii
ABSTRACT.....	vi
LIST OF TABLES	vii
LIST OF FIGURES.....	viii
CHAPTER	
1. INTRODUCTION	1
1.1 Introduction.....	1
2. LITERATURE REVIEW	3
2.1 Introduction to User Interfaces	4
2.2 User Interfaces are difficult to design.....	7
2.3 Portability.....	9
2.4 Portable screen handling and curses.....	15
2.5 Software architecture and user interfaces	15
2.6 Guidelines for user interface software design.....	20
2.7 Survey on user interface programming	28
2.8 Guidelines for cross platform software development.....	31
2.9 State of art in user interfaces.....	33
2.10 Research issues in cross platform user interfaces	34

3. EXISTING INDUSTRY STANDARDS IN CROSS PLATFORM USER INTERFACES/ PLATFORMS	36
3.1 Microsoft Foundation Classes (MFC) and Visual C++	36
3.2 SUIT (SIMPLE USER INTERFACE TOOLKIT).....	37
3.3 Other Products	37
4. EXISTING TECHNIQUES AND METHODOLOGY	40
4.1 Current Existing Eechniques	40
5. CROSS PLATFORM USER INTERFACES	44
5.1 Abstract Factory	45
5.2 Bridge	47
6. SOFTWARE ARCHITECTURE FOR CROSS PLATFORM USER INTERFACES	50
6.1 Introduction.....	50
6.2 Models for describing user interface software architecture.....	52
6.3 Case studies of Software Architecture	56
6.4 Software architecture and steps necessary to convert the built GUIlibrary into a cross platform library.....	94
7. RESULTS	104
7.1 The developed library	104
7.2 SUIT and non-virtual widgets	108
8. CONCLUSIONS AND RESEARCH DIRECTIONS	110
8.1 Conclusions.....	110
8.2 Research directions	112

BIBLIOGRAPHY	114
APPENDIX: STRUCTURE OF A ZAPP APPLICATION.....	118

ABSTRACT

This research is a study of user interfaces, and in particular, cross platform user interfaces. The problems associated with portability and the challenges and difficulties associated with developing user interfaces are examined. The models used for describing user interfaces architectures are evaluated and compared. The techniques used to build cross platform user interfaces and the existing user interface industry standards are examined and the requirements and the characteristics of cross platform user interface architectures are examined in great detail. Also, the steps necessary to convert a built GUI library into a cross platform library are defined.

LIST OF TABLES

2.1.	Byte ordering in different processors	12
2.2.	Features of PIGUI currently available in the market.....	13
2.3.	Hardware and software combinations supported for PIGUI	14
6.1.	Comparison of architectures on different platforms.....	103
7.1.	Characteristics of the architecture for the GUI library to be improved further.....	108

LIST OF FIGURES

2.1.	Venn diagram representing an interactive computer based system	4
2.2.	UIMS model	17
2.3.	Overall process for interface	21
2.4.	The Scoping phase	22
2.5.	The functional specification phase	23
2.6.	The design phase	26
2.7.	The development phase	27
2.8.	Testing and Implementation.....	28
2.9.	Percent of time devoted to user interface during the various phases of system development	29
2.10.	Comparison of average percentage of code devoted to user interfaces.....	30
2.11.	Comparison of implementation times devoted to user Interface for projects with different levels of use	30
2.12.	Windows processing	31
2.13.	OS/2 processing	31
2.14.	DOS processing	31

3.1.	MainWin architecture	38
3.2.	zApp architecture	39
4.1.	SUIT architecture	41
5.1.	Abstract factory	46
5.2.	Portable window abstraction.....	47
5.3.	Bridge pattern.....	48
6.1.	Arch model	52
6.2.	An elementary interactive object.....	54
6.3.	PAC structure.....	56
6.4.	Serpent architecture.....	57
6.5.	Chiron architecture with functional roles annotated.....	60
6.6.	Garnet architecture	62
6.7.	Garnet architecture with functional roles annotated.....	63
6.8.	CDocument and CView hierarchy model.....	67
6.9.	A Visual C++ application.....	68
6.10.	MFC architecture	74
6.11.	MFC thread of execution.....	75

6.12. Layered software for portability (SUIT)	80
6.13. SUIT architecture	83
6.14. A SUIT application	84
6.15. SRGP multiplexing scheme	87
6.16. SUIT layered model	88
6.17. zApp Interface Data flow	90
6.18. zApp architecture	91
6.19. zApp class hierarchy	92
6.20. Include files for the built library.....	95
6.21. Windows application.....	98
7.1. DOS Demo	105
7.2. Graphics Demo	106
7.3. Windows Demo	107

CHAPTER 1

INTRODUCTION

1.1 Introduction

A platform may be defined as a combination of hardware, operating system and windowing system [5]. With the large number of computing platforms, it is very difficult for a company or individual to make choices about a computing platform at a particular time. Even if the platform does not change, the view of it eventually will. Applications commonly have to be ported to the new environments. Therefore the ease of portability is important. Portability determines the degree to which a program or software can be moved from one computer to another.

When programs written for a particular platform cannot produce identical results from identical data written for a particular platform when compiled across different platforms, due to portability problems, steps must be taken to get the desired results.

When a program with a user interface is ported from one platform to another, not only the application part of the source code has to be ported, the user interface part of the code has to be rewritten for the new platform. This becomes a very tedious task for a programmer to rewrite the user interface code for each platform.

The goal of the current proposal is to examine user interfaces and cross platform user interfaces. The research is divided into two parts. First, the software architecture of cross platform user interfaces will be examined in

detail. Secondly, a GUI library will be built on DOS Text, DOS Graphics and MS-Windows. The steps necessary to convert this GUI library into a cross platform library will be examined.

Chapter 2 deals with literature survey where portability problems and guidelines for user interface software development are discussed. Chapter 3 examines some standards in cross platform user interfaces in the industry. Chapter 4 discusses the techniques used to develop cross platform user interfaces. I will be discussing a few design patterns used for cross platform object oriented programming in Chapter 5. Chapter 6 deals with models for describing software architecture and a few case studies of cross platform user interfaces. The steps necessary to convert the built GUI library into a cross platform library are examined here.

CHAPTER 2

LITERATURE REVIEW

The user interface (UI) of a computer program may be defined as that part of the program that handles the output to the display and the input from the person using the program [21]. Tools used to develop user interfaces have been called various names over the years, with the most popular being User Interface Management Systems (UIMS), Toolkits, User Interface Development Environments, Interface Builders, Interface Development Tools and Application Frameworks.

We can classify the people involved with user interface software into four main categories.

1. The person who uses the resulting program is called the end user or user.
2. The person who creates the user interface of the program is called the user interface designer or just designer.
3. Working with the user interface designer is the person who writes the software for the rest of the application. He or she is called the application designer.
4. The designer may use special user interface tools which are provided to help create user interfaces. These tools are created by the tool creator.

With a uniform user interface across all platforms, the user can access the application without the knowledge of the underlying hardware and operating system changes to which the application has been ported to.

The main goal of cross platform user interfaces is to generate the user interface across all platforms the programmer wishes to port his application to, with one set of source code. When this is achieved, the programmer has to just interlace the application code within the user interface code to generate the desired application, when compiled separately across all platforms.

Cross platform user interfaces are also known by the name of “Platform Independent Graphical User Interfaces (PIGUI).”

2.1 Introduction to User Interfaces

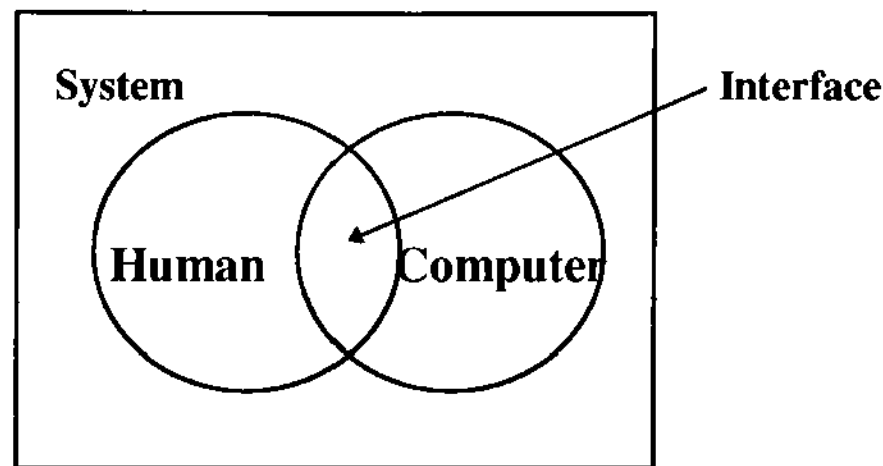


Figure 2.1. Venn diagram representing an interactive computer based system.

If we consider a Venn diagram representing an interactive computer based system, the main components include the Human and the Computer (Figure 2.1). The intersection between these two sets defines the area of Human Computer Interface and hence all the studies related to Human Computer Interactions (HCI). The study of user interface software is one of the key areas in HCI.

Considering the human system, it is flexible and adaptable. The difficult part here is the design of the interface for user.

Some of the general principles of User Interface Design are [6]:

1. User Compatibility

The designer of user interfaces must be familiar with basic cognitive psychology or an understanding of the general strengths and weaknesses of the human mind.

2. Product Compatibility

Users of the new systems must be able to use it with the working knowledge of the previous system. If this is not accomplished, the user has to start all over again.

3. Task Compatibility

The structure and flow of a system should match and support the task that is being carried out. It is better to organize user interfaces by the task type sometimes.

4. Work Flow Compatibility

Systems should also be organized to facilitate transitions between tasks. User interfaces can support switching between tasks.

5. Consistency

Consistency is related to compatibility. It refers to similarity within a product.

6. Familiarity

An important characteristic that can greatly facilitate the learning of a new interface is familiarity, concepts, terminology and spatial arrangements that the user is already familiar, which can be incorporated into the interface.

7. Simplicity

One goal that should be kept in mind while designing user interfaces is simplicity. The user interface must be easy to navigate for a novice. The main idea is to keep a novice away from the full complexity of the underlying system.

8. Direct manipulation

A direct manipulation interface is one in which users directly perform actions on visible objects.

9. Control

The interface must be designed in such a way that the user has to have control over the machine. The designer should be careful enough to see that the user is not frustrated or demoralized and present a good interface.

10. WYSIWYG

It is an acronym for “What You See Is What You Get.” It generally refers to the degree of one to one correspondence between information displayed on the screen and the information displayed on printed output or stored in files.

11. Responsiveness

The computer should respond immediately to its users input. What the computer does internally must be invisible to the user.

12. Invisible Technology

The user must know as little as possible about the technical details of how the system is implemented and operates.

13. Robustness

A system should tolerate common and unavoidable human error. System crashes should be minimized and simple to understand recovery measures should be minimized.

14. Protection

The user should be protected against catastrophic results of common human error.

15. Ease of learning and ease of use

Systems should be easy to learn and use.

2.2 User Interfaces are difficult to design

Some of the reasons why user interfaces are difficult to design are [20]:

1. Designers have difficulty learning the user's tasks

Surveys of software in general show that the deep application-specific knowledge required to build large, complex systems is held by very few developers and is hard to acquire.

2. Inherent complexity of tasks and applications

The increased complexity comes from complex requirements in the domain itself. One way to try to overcome complexity is to use metaphor that exploits the user's prior knowledge by making interface objects seem like objects that the user is familiar with.

3. The variety of different aspects and requirements

All designs involves tradeoffs, but it seems that user interface design involves a much larger number of concerns, and they are the purview of

widely different disciplines. User interface design should include considerations about:

a. Standards

It mainly refers to look and feel factors.

b. Graphic design

It is the graphical presentation including layout, colors, icon design, text fonts, etc.

c. Documentation, messages and help text

Rewriting help messages, prompts and documentation to increase the quality has significantly more impact on the usability of the system than varying the interface style.

d. Internationalization

Internationalizing an interface is much more difficult than simply translating the text strings and may include different month, date and time formats, new input methods, redesigned layouts, difficult color schemes and view icons.

e. Performance

Users will not tolerate interfaces that perform too slowly.

f. High level and low level details.

Not only has the overall model has to be correct, each low level detail must also be perfected.

4. Theories and Guidelines are not sufficient

There are many methodologies for User Interface design. Most sources put consistency at the top of their guidelines.

5. Difficulty of Iterative Design

Due to all difficulties described above, all HCI professionals and HCI methodologies recommend iterative design. The problem with iterative design is that the new version may be worse than the older one.

6. Implementation issues

The main reason why user interfaces are difficult to implement is the need to use iterative design which adopts the conventional software engineering “waterfall” model approach to software design where the user interface is fully specified, then implemented and later tested is inadequate. Instead, specification, implementation and testing must be intertwined.

2.3 Portability

The two major reasons due to which portability is necessary are

1. Hardware Problems

Hardware Problems arise due to the incompatibility of the underlying hardware on a host machine while software problems arise due to the incompatibility of the software running across different platforms.

Hardware problems can be due to:

a. Instruction sets

Different host machines are built around different processors and hence have different instruction sets. Any code that conforms to or uses these instruction sets (in assembly language) will run into problems when being ported.

b. Data Representation

Computers have different types of internal representation of negative numbers. They may follow one's complement or two's complement arithmetic. Also, a processor may be big-endian or small-endian.

c. CPU speed

If speed is a factor in an application, then the CPU speed matters play an important role. As a general rule, the faster the processor, the better it is for an application.

2. Software Problems

a. Compilers:

Standards which are defined may not yet be available on compilers on different platforms. Some areas in definition may be deliberately or accidentally left vague. Some definitions may be interpreted differently by different people.

b. Operating systems:

The main problems for portability are most of the times in the operating systems. The operating systems can be single user, multi-user, real-time, batch or combination of the last three. Although the operating system conceals the details of the hardware from the user, how the operating system does the housekeeping is not of relevance although how it affects the user is to be examined.

Some of the regions of incompatibilities may be in the following areas.

1. Filenames:

There is a limit on the number of characters in a filename and its extensions. This may differ from one operating system to another.

2. Quotas:

Some operating systems allocating space for files may impose quotas for the users.

There can be programs written for one kind of platform processor which can run on multiple platforms. Windows NT is a typical example for this. Also, programs written for Windows 3.1 (Win16 or Win32) can be ported to Windows NT and vice versa. The main thing to be noted here is that Windows 3.1 is confined to Intel Platforms whereas Windows NT can run on multiple platforms. Microsoft Windows NT is a typical example of a portable operating system and demonstrates processor portability.

Some 16 bit applications do not run on non-Intel processors because they use 386 level (or above) machine instructions. Since Windows NT emulates only 286 level machine instructions, a 16 bit binary that contains 386 level or above instructions will generate unsupported instruction exceptions when run on non-Intel platforms [27].

When applications are ported from one platform to another, the underlying processor plays a very important role. There are many problems as far as processors are concerned. The main issue here is the byte order the processor follows. Processors may be big-endian or small-endian.

Big endian processors store the most significant byte at the lowest address while the small endian processors store the least significant byte at the lowest address.

This determines padding in structures used in our programs. The following table (Table 2.1) gives a few CPUs and their byte ordering.

Table 2.1: Byte ordering in different processors

CPU	Variable	Byte ordering
Intel 80x86	int long	b1,b0 b3,b2,b1,b0
Motorola 680X0	int long	b0,b1 b0,b1,b2,b3
DEC Alpha	int long	b0,b1 b0,b1,b2,b3

If the binary files on one platform have to be used on another platform, a byte ordering program has to read the bytes and convert it.

There are a lot of PIGUIs available in the markets today. They can have GUI developmental solutions both for C and C++.

Table 2.2 gives a number of PIGUIs and the features associated with it [26].

Table 2.2: Features of PIGUI
currently available in the market

Product	Type	Eval	Source	Royalty	Distrib	Lang	Builder
Appware				no			
Aspect		30		no		C	yes
Views	layer		free	no		C++	yes
CLIM						Lisp	
CommonV	layer		yes	no		C++	
DCLAP		free	free			C/C++	
Galaxy	emu			no		C/C++	yes
Guild				no		C	yes
JAM	layer		yes	no		C	yes
libWxm	API						
MAINWin	API	30	no			C/C++	\$199
Meneur							
MEWEL	API			no		C	
ObViews	layer			no		C++	
OI	emu			no	8K	C/C++	yes
OpenUI	layer			no		C	yes
PSM				yes			no
ScrMach	layer	30		no		Ads	free
StarView	layer	30	yes	no		C++	yes
SUIT		free	free			C	
VisWork	emu	30	100K	yes		Smalltalk	yes
Wind/U	layer	30		no	19.5K/y	C/C++	
WNDX	layer	30		no		C	yes
wxWindows		-	free	no		C++	
XVT	layer		yes	no		C/C++	yes
zApp	layer	60	free	no		C++	\$499
Zinc	layer	60	free	no	-	C++	yes

Type: Emulated, layered or API emulated.

-: package does not support feature.

Blackboxes: unknown

blank:other conditions apply

Table 2.3 gives the hardware and software combinations supported for all PIGUI.

Table 2.3: Hardware and software combinations supported for all PIGUI

Appware Aspect	C++ Views	CLIM	Common V	DCLAP	Galaxy	Guido	JAM	libWxm	MAINWin	Menue	MEWEL	ObViews	OL	OpenUI	PSM	SetMach	StarView	SUI	VisWorks	WindU	WINDX	wxWindows	XVT	zApp	Zinc		
p	y	-																								DOS	Graphics
-	y	p																								DOS	Text
																										DOS Extender	16-bit
																										DOS Extender	32-bit
-	y	-																								ASCII Text	
y	y	y		y	y	-	y	e	y	e	e	y	y	y	y	y	y	y	y	y	e	y	y	y	y	MS-Windows 3.x	Win16
																										MS-Windows 3.x	Win32
y	y	p		p	y	y	y																			Macintosh	
																										Macintosh	PowerPC
p		y		y		y		y	-	y	-	y	y	y	p	-	y	y	y	y						OS/2	
p		p			y	y			y	p	y	y	p			p	y			y						Win NT Win32	80x86
p		p			p				y	p		y	-			p	y									Windows NT	DECAXP
p		-			p				y	p						p	p									Windows NT	MIPS
-	y	-		p		y							y	-			y	y	y	y						SunOS	OpenLook
y	y	y			y	y	p	y						y			y	y	y	y	y	y	y	y	y	SunOS	Motif
		p							y											y						Sun Solaris 2.x	Motif
p	y	-							p	-				y		y	-									ISC	Motif
																										AT&T Sys 3000	Motif
p	y	y							p	-			y	y	y	y	p									SCO Unix/Xenix	Motif
p	y	-			y			y	p	-			y	-			p				p	y	y			MIPS	Motif
y	y	y			y			y		y	-		y	y		y	p	y	y	y	y	y	y	y	y	HP 9000/HPUX	Motif
p	y	p			y			y		y	-		y	y		y	y	y	y	y	y	y	y	y	y	IBM RS6000/AIX	Motif
																										DEC AXP OSF1	Motif
																										DECAXPOpenVMS	Motif
y		-			y				p	-			y	y		y	p		y			y	y	-		DEC Ultrix	Motif
p	y	p			y				p	-			y	y		y	p				p		y	-	p	DEC VaxVMS	
p	y	-			y			y		y	-		y	-		y	p	y		y	y	y	y	y	y	Silicon Graphics	
-		-																								Pyramid	

e:API emulated by the software.

y: verified by user report or claimed to work in vendor literature.

p: planned, but not yet in beta test.

black boxes: unknown.

-: Package does not support a feature or platform.

2.4 Portable screen handling and curses

Curses is a library of screen and window management routines. It is modeled after the UNIX curses and the DOS ncurses libraries. Normally, programs written for DOS should be easily portable to UNIX and vice versa [8].

To use the routines, the functions `initscr( )` must be called. This creates two windows: `stdscr` and `curscr`. `Stdscr` is the default window for the user to make changes on and `curscr` reflects the current contents of the physical display screen.

Hence programs written to curses library can be ported to all platforms that it is written on like DOS, UNIX, VAX, etc. Curses form one more cross platform user interface porting methods.

2.5 Software architecture and user interfaces

2.5.1 Introduction

Software architecture is the study of large-scale structure and performance of software systems [16]. Software architecture is a topic of growing concern within both research and industrial circles. User interface architectures are typical of architectures in other areas since they are primarily motivated by organizational or life cycle qualities. Examples of such qualities are maintainability, portability, modularity, reusability and so forth.

The architectural design of a software system can be described from (at least) three perspectives. The functional partitioning of its domain of interest,

its structure, and the allocation of domain function to that structure. According to Garlan:

The software level of design addresses structural issues such as gross organization and global control structure; protocols for communication, synchronization and data access; assignment of functionality to design elements, composition of components, scaling performance, and selection among design alternatives.[15, p.6]

“Architectural design is the activity of partitioning the requirements to software subsystems”[15, p.6].

Software architecture alludes to two important characteristics of a computer program:

1. The hierarchical structure of procedural components and
2. Structure of data.

Software architecture is derived through a partitioning process that relates elements of a software solution to parts of a real-world problem implicitly defined during requirements analysis [25]. “System architecture shall describe the internal structure of the system. The segments ...shall be identified and their purpose summarized. The relationships among the segments...shall be described”[15, p.6].

Those architectures that do not describe an actual implementation usually serve as reference architectures for the entire domain of user interfaces, that is they are sufficiently abstract so as to apply to a wide variety of implementations [15].

2.5.2 Models for describing user interface software architecture

There are a number of models that have been proposed for user interface architectures. A few of them are [15].

2.5.2.1 UIMS model

The figure below (Figure 2.2) illustrates the UIMS model.

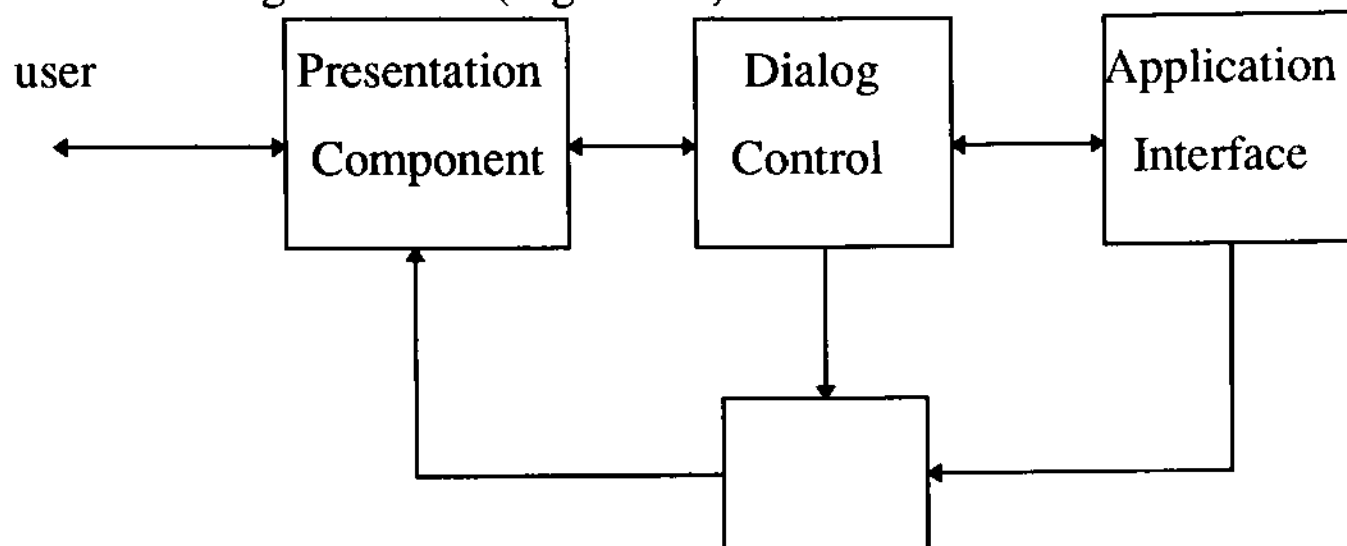


Figure 2.2. UIMS model.

The three main components of this model are

1. Presentation,
2. Dialog,
3. Application interface.

The dialog component is least familiar. It is essentially an organizer of traffic between the presentation components. It is responsible for

- Maintaining a correspondence between objects from the application domain and interface objects from presentation domain.
- Maintaining relationships among the interface objects.
- Maintaining a dynamic representation of the state of the dialog between the other two components.

- Defining two communication protocols, one suitable for the functional core and second for presentation.

The Seeheim model does not provide a complete description of a user interface architecture. The model was presented as a logical model and so fits our description of a functional partitioning.

2.5.2.2 The Arch/Slinky metamodel

The Arch Slinky metamodel of user interface architectures is an extension of the basic Seeheim model and identifies the following basic components of the user interface software.

- Functional core (FC)

This component performs the data manipulation and other domain oriented functions. It is these functions that the user interface is exposing to the user. This is also commonly called the Domain Specific Component or simply the Application.

- Functional Core Adapter (FCA)

This component aggregates domain specific data into higher-level structures, performs semantic checks on data and triggers domain initiated dialog tasks. The FCA is similar to the application interface in the Seeheim model.

- Dialog (D)

This component mediates between the domain specific and presentation specific portions of a user interface, performing data mapping as

necessary. It ensures consistency (possibly among multiple means of data) and controls task sequencing.

- Logical Interaction (LI)

This component provides a set of toolkit independent objects (sometimes called virtual objects) to the dialog component.

- Physical Interaction (PI)

This component implements the physical interaction between the user and the computer. It is this component which deals with input and output devices. This is also called the Interaction Toolkit Component.

The Arch/Slinky model is also a functional partitioning, but is both more explicit about its states as such, and more fine grained than the Seeheim model.

2.5.2.3 PAC model

The PAC conceptual model for interactive systems is understood to consist of an interaction system represented as a collection of Presentation-Abstraction-Control (PAC) agents which are related in a somewhat top down hierarchical structure.

Each PAC agent triple represents three tightly connected components. The abstraction component contains some application relevant piece of the system, the presentation component some interaction relevant piece and the control component maintains a presentation-abstraction correspondence within the agent.

2.6 Guidelines for user interface software design

User interface software design cannot follow top down approach in design. The design must be more iterative. Iteration is employed in both software engineering and user interface development because of our inability to account for all details at once (Figure 2.3).

Software engineering of the user interface can be top down in the sense of matching the user interface software to the interface software requirements as produced in each iteration of the abstract iteration design.

The goal of software engineering is to achieve correct and complete functionality. Iteration in software design is therefore used to ensure that requirements are eventually correct and that data structures and functionality algorithms are correct and complete implementations fulfilling the requirements.

Formative evaluation (testing as a part of development process) is done to ensure that the functionality is as complete as possible and that software engineering is working correctly [11].

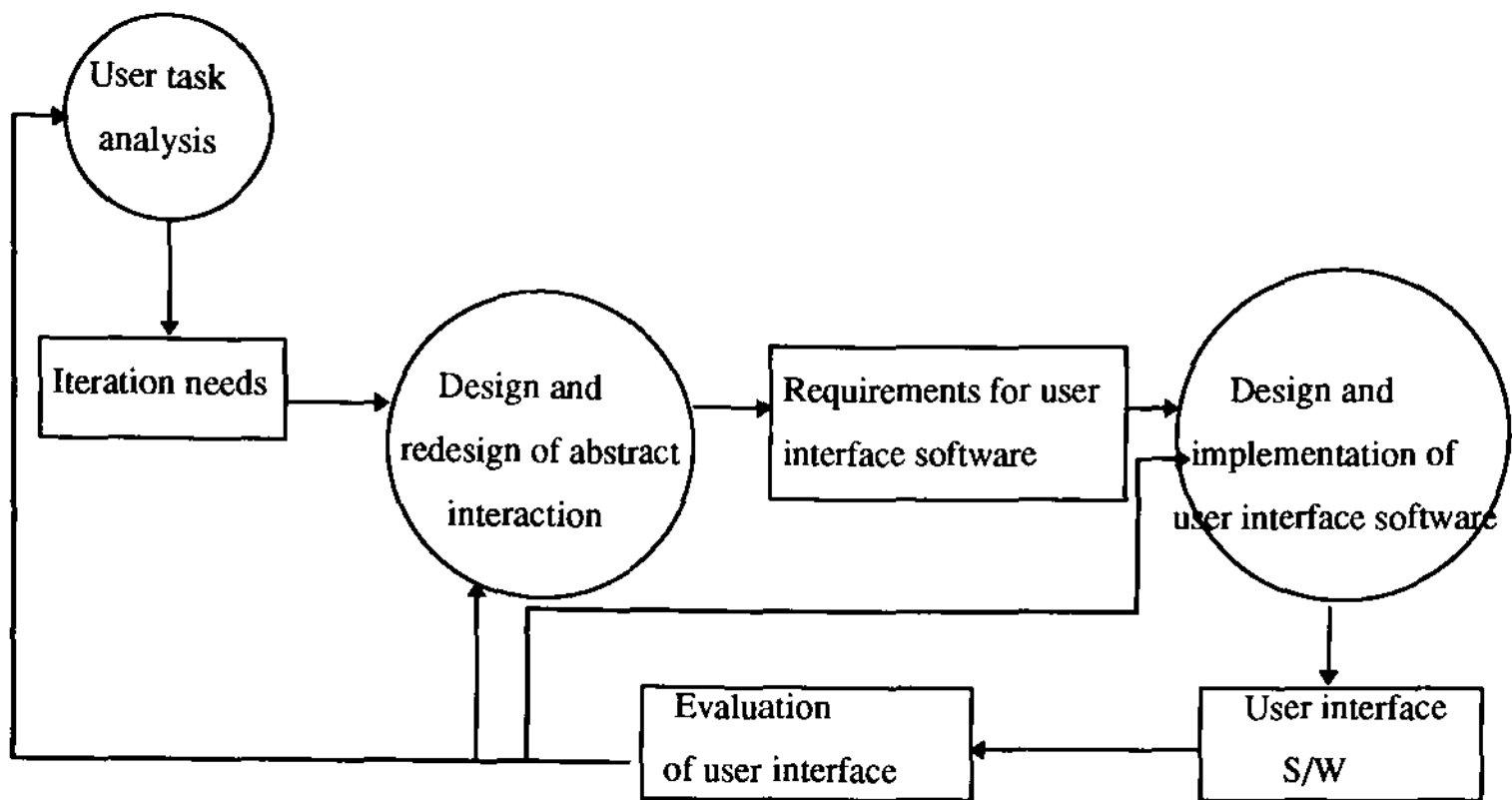


Figure 2.3. Overall process for interface must be iterative and self correcting, and cannot be top down.

The methodology for user interface design consists of a set of user interface design tasks applied in a particular order at specified points in the overall development life cycle. All tasks in a typical system development life cycle are presented in the order in which they would be carried out.

Each task is represented as a box enclosing the task name [6].

2.6.1 Scoping Phase

2.6.1.1 Project plan

A particular scope and schedule for each of the remaining tasks is determined and integrated with the overall project plan (refer to Figure 2.4).

2.6.1.2 User profile

A description of the user characteristics relevant to the user interface design is obtained.

2.6.1.3 Hardware and software definition

The user interface made possible by hardware and software platform chosen for application are documented.

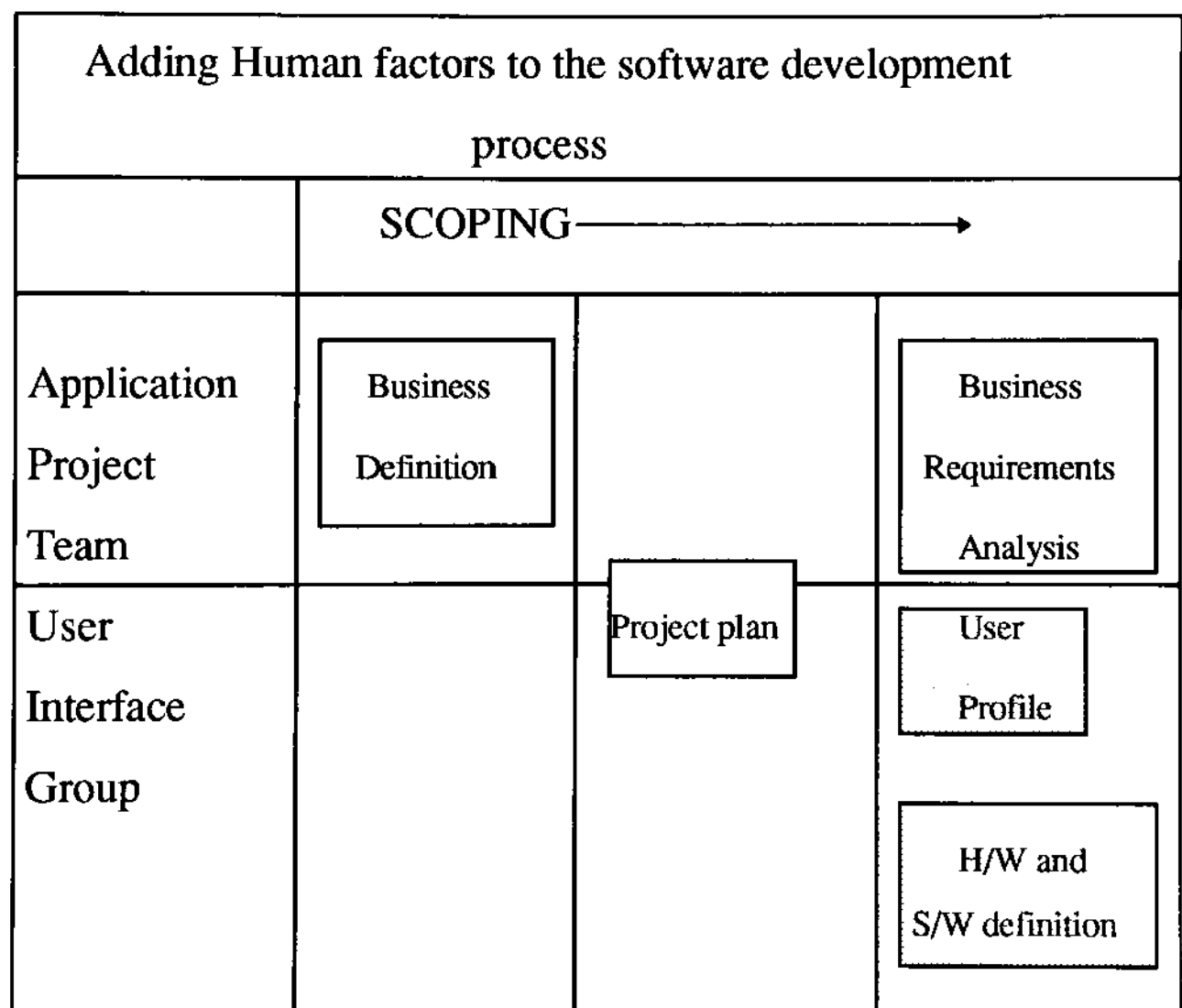


Figure 2.4. The scoping phase.

2.6.2 Functional Specification

2.6.2.1 Task Analysis

A study of the user tasks, work flow patterns, and conceptual frameworks is made, and this drives a very high level conceptual design of user interface (refer to Figure 2.5).

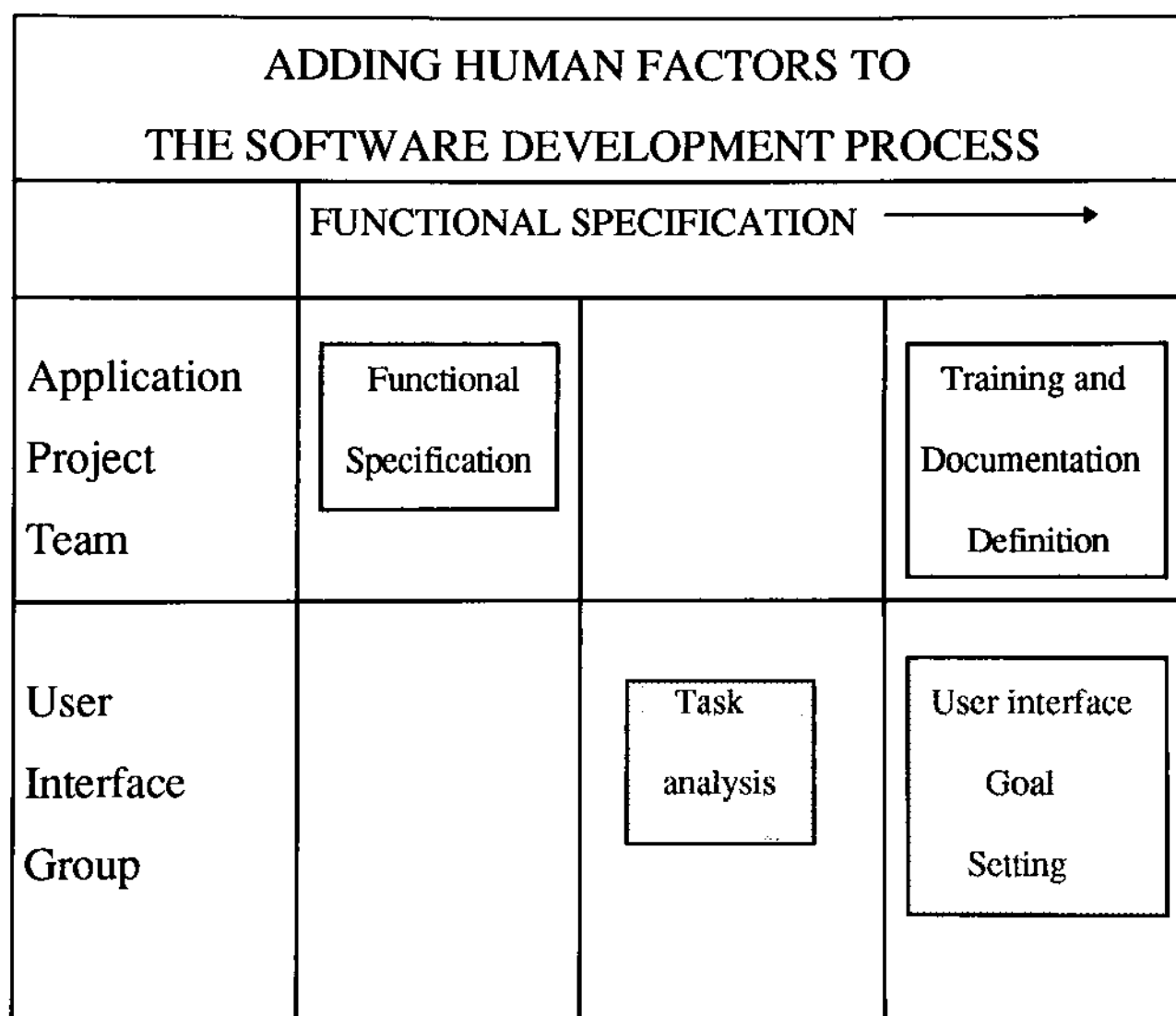


Figure 2.5. The functional specification phase.

2.6.2.2 User interface goal setting

Specific quantitative goals defining minimal acceptable user performance are defined. These focus later efforts and form the basis for the test plan developed later.

2.6.2.3 Training and documentation definition

The various forms of hard-copy documentation, training materials, tutorials, and other on-line user aids that will be provided with the application are decided on. Key staff members who will be responsible for developing these materials and facilities are identified and participate in all later tasks.

2.6.3 Design

2.6.3.1 User interface mockup

Based on the high level conceptual design generated in the task analysis and any existing corporate user interface standards, initial design ideas are generated and implemented in story board form or on a rapid prototyping tool. Several alternative designs may be mocked up (refer Figure 2.6).

2.6.3.2 Style guide

The high level design (overall structure, specific implementation of corporate user interface standards and navigational pathways) determined in

the previous two tasks are documented in a user interface specification called style guide.

2.6.3.3 Detailed user interface design

Detailed user interface design (for example, screen design and error message design) is conducted and documented.

2.6.3.4 User interface prototype

The detailed design is implemented in full prototype form on a rapid prototyping tool.

2.6.3.5 Prototype user interface test plan

A plan is developed for a testing procedure that will reveal any major flaws in the initial design.

2.6.3.6 Prototype user interface testing

High level testing of the user interface prototype is conducted iteratively until all major identified problems are eliminated.

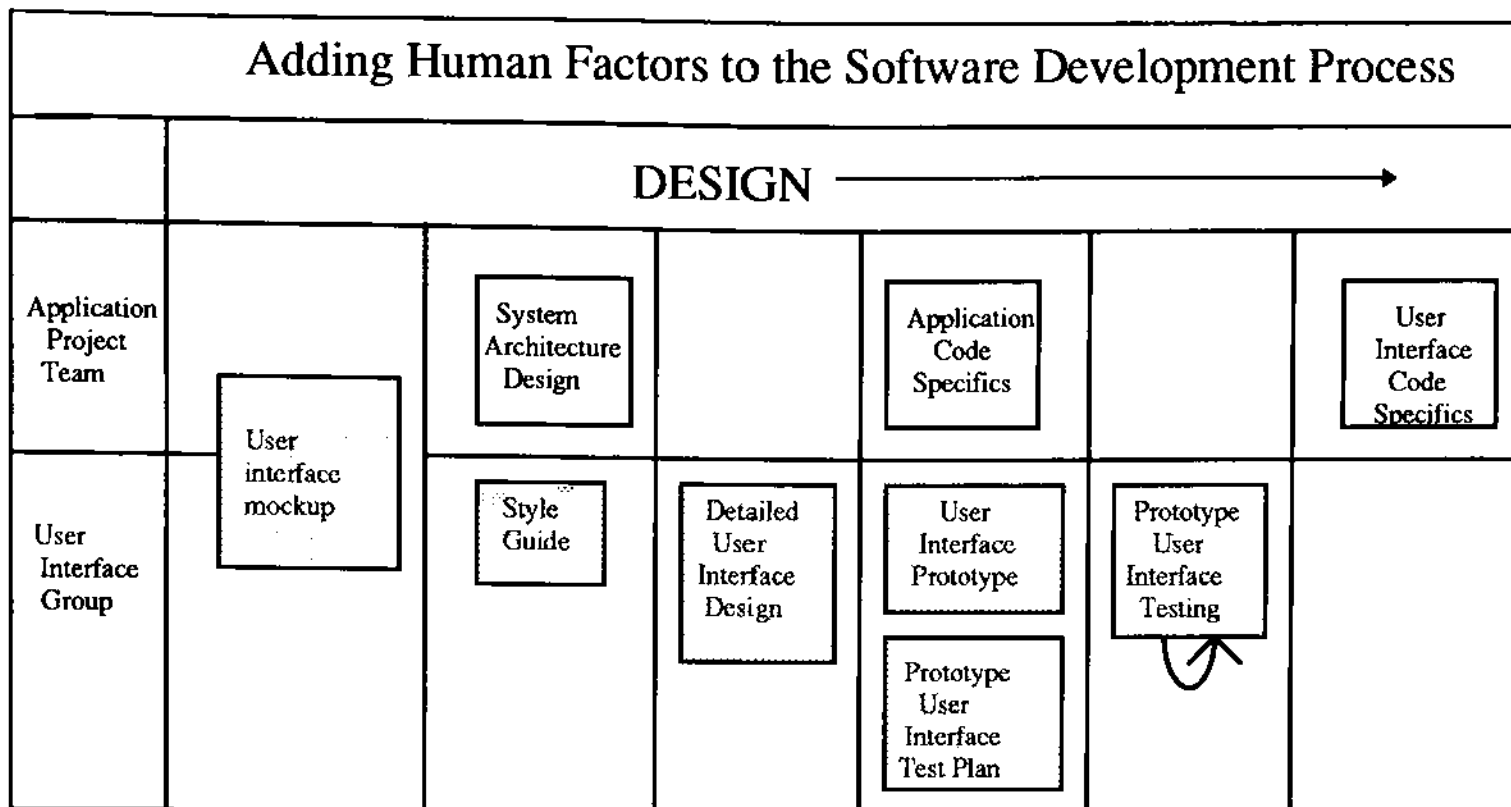


Figure 2.6. The design phase.

2.6.4 Development

2.6.4.1 Develop training and documentation

User manuals, on-line help, tutorials and training materials are designed and developed (refer to Figure 2.7).

2.6.4.2 User interface test plan

A plan for formal testing of the full system against the ease of use and learning goals identified during the definition phase is designed.

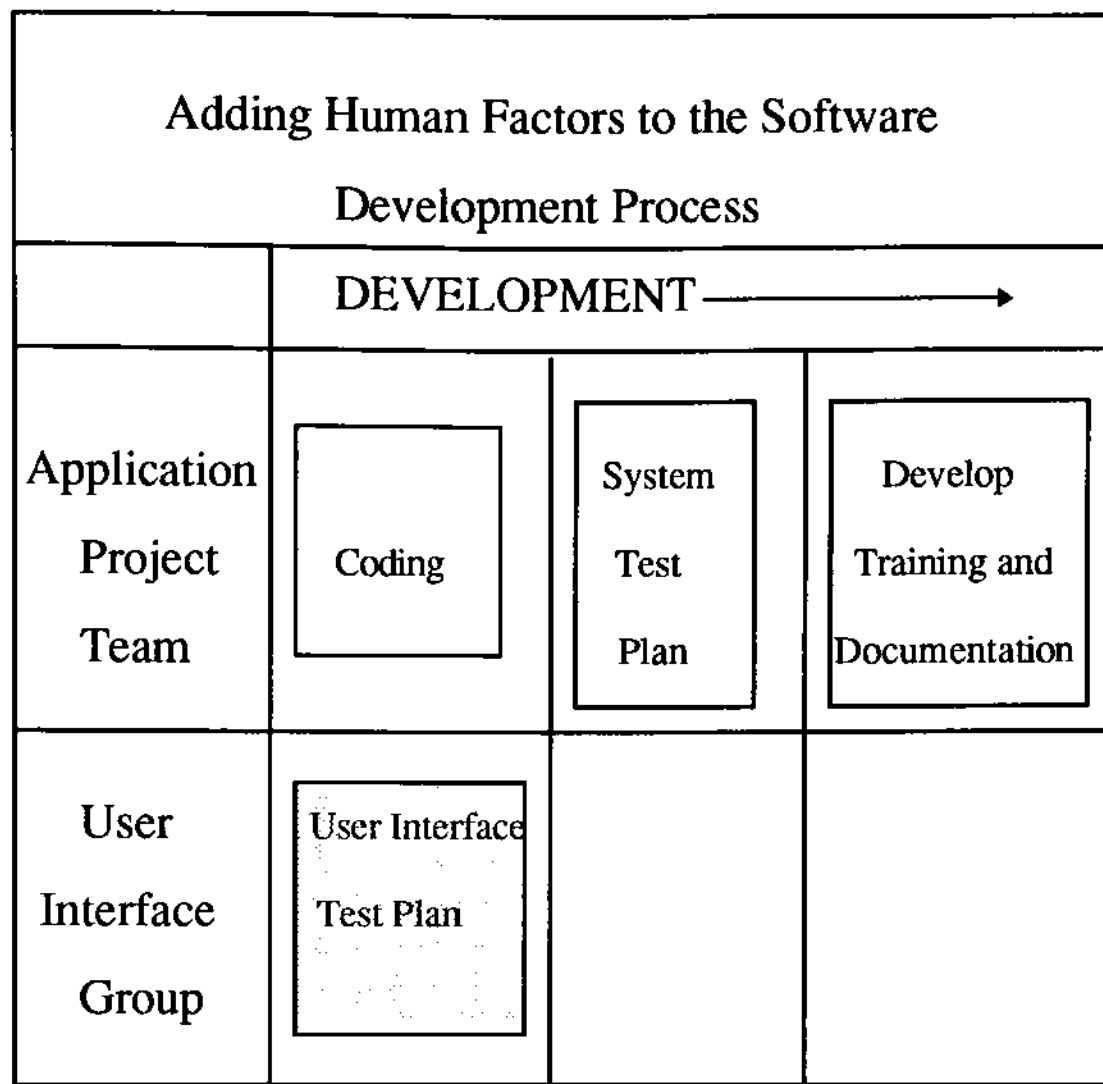


Figure 2.7. The development phase.

2.6.5 Testing/Implementation

2.6.5.1 User interface testing

Testing and redesign are conducted iteratively until the ease of use and ease of learning goals are met (refer to Figure 2.8).

2.6.5.2 User interface evaluation

After the application has been in production for three months or more, feedback is gathered to feed into enhancement design and/or application design.

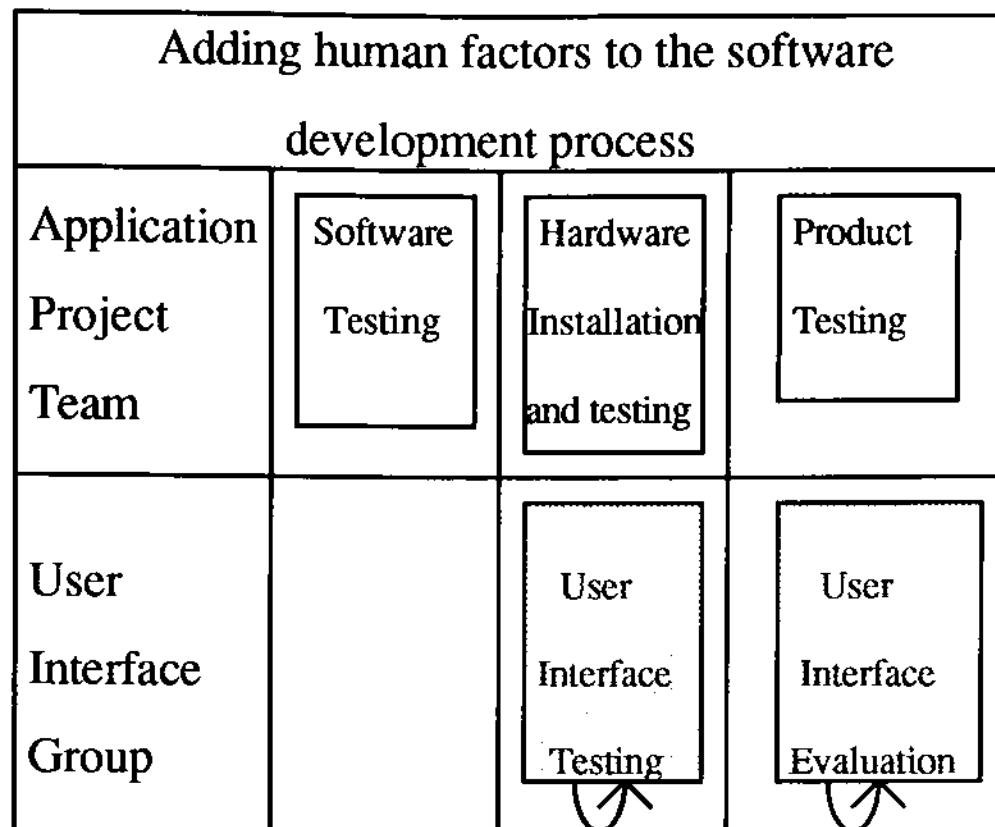


Figure 2.8. Testing and implementation.

2.7 Survey on User Interface Programming

A survey conducted by Myers [19] revealed that of the average percentage of time devoted to the user interface during the various phases of system development, 44.8% was devoted to design of user interfaces, 50.1% to implementation and 37% to maintenance (Figure 2.9).

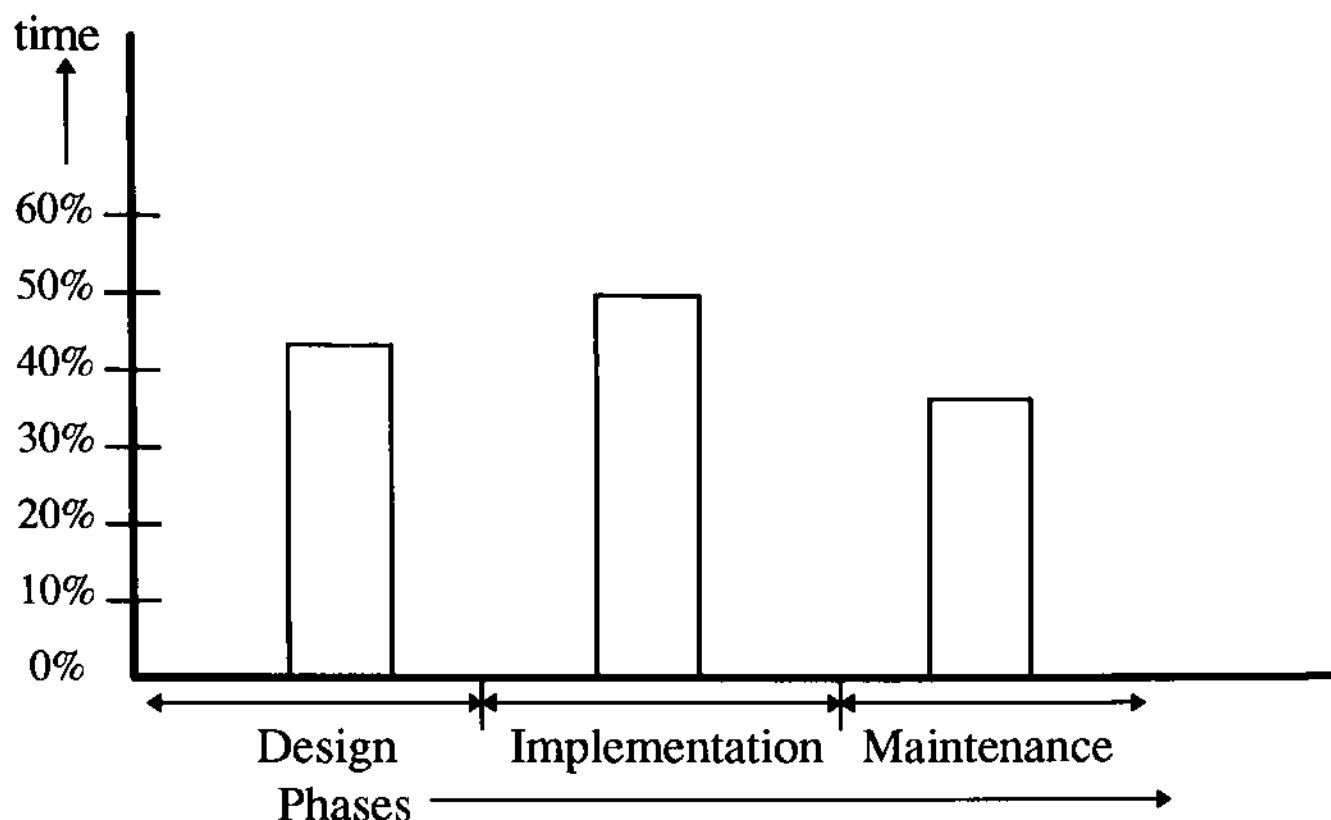


Figure 2.9. The average percent of the time devoted to the user interface during the various phases of the system development (N=63 for design, N=63 for implementation and N=42 for maintenance, N=number of respondents).

A comparison of average percentage of code devoted to user interface projects revealed that 45.2% used tools, 57% used toolkits and 40.6% used UIMS or Builders (Figure 2.10).

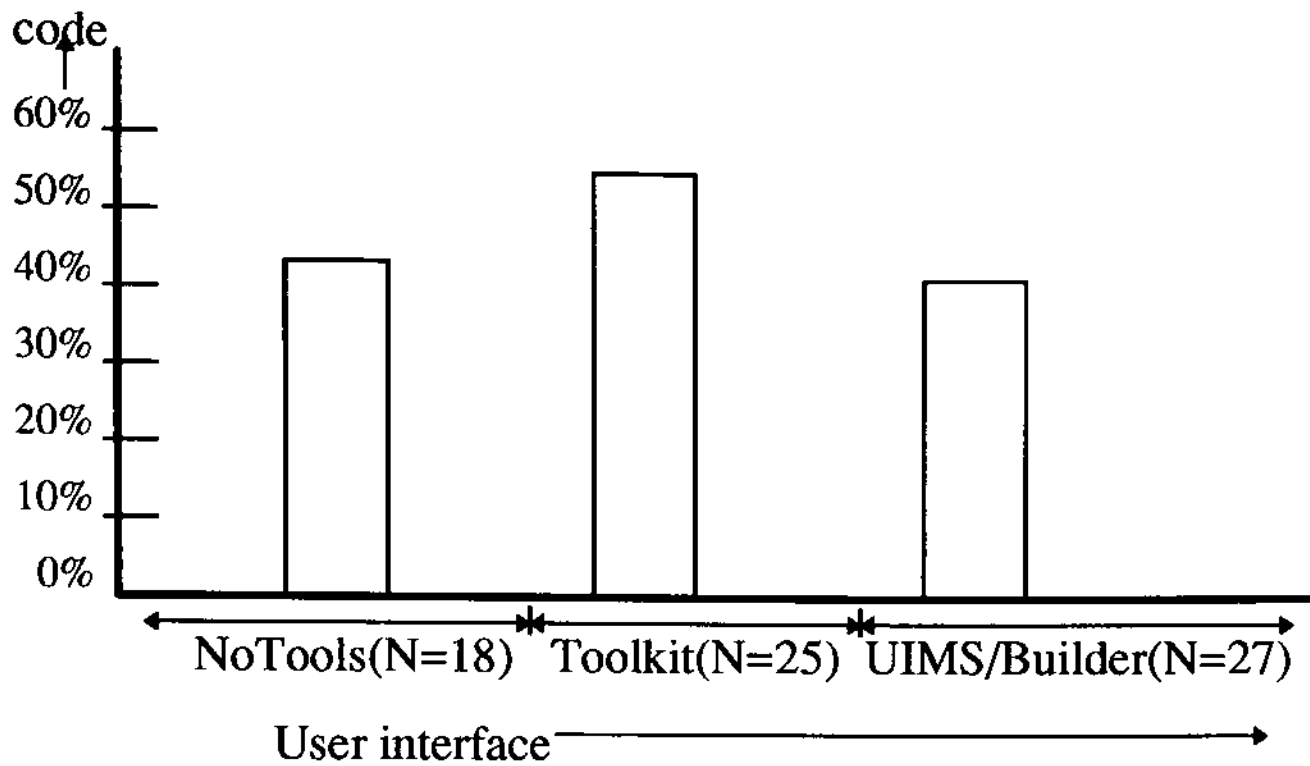


Figure 2.10. Comparison of average percentage of code devoted to user interface.

A comparison of implementation times revealed that 44.0% used tools, 64.9% used toolkits and 41.2% used UIMS or Builders (Figure 2.11).

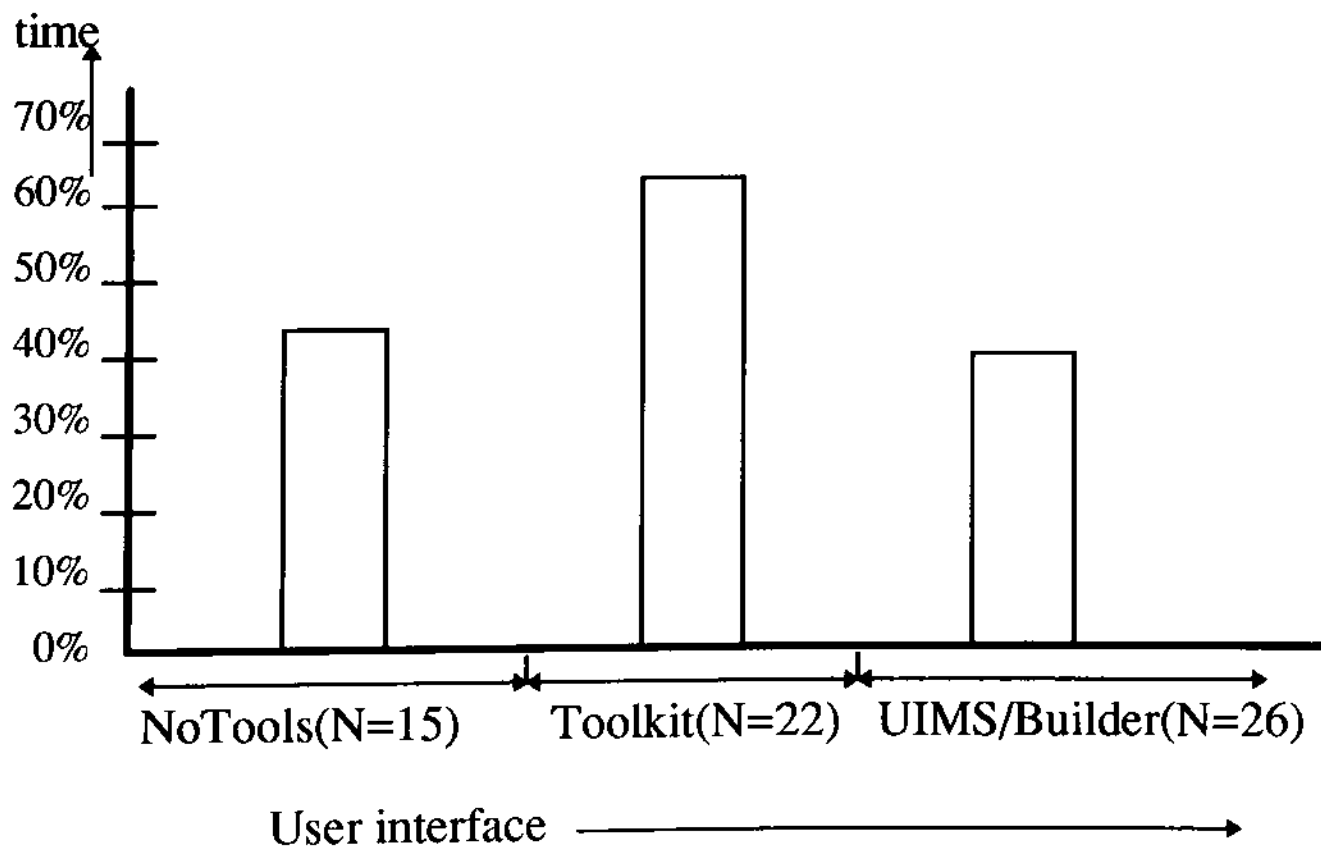


Figure 2.11. Comparison of implementation times devoted to user interface for projects with different levels of use.

Regarding tools used for user interface development, 74% used tools of some sort. For many projects (34%), this consisted of a toolkit providing a library of user interface components.

2.8 Guidelines for cross platform software development

Cross platform user interface development (PIGUI) is concerned mainly with creating user interfaces on different platforms using just one set of source code [10].

Consider Figures 2.12 to 2.14 below.

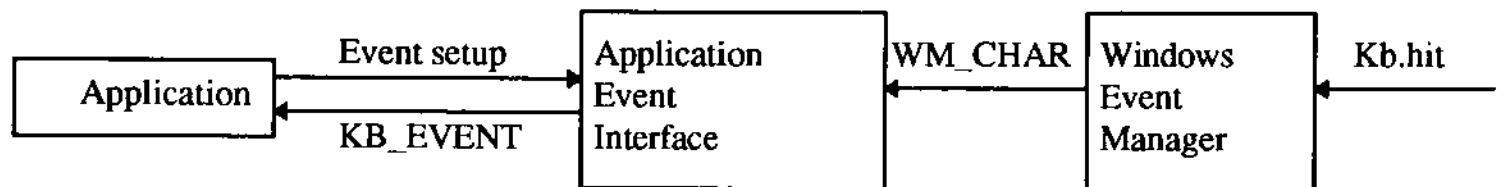


Figure 2.12: Windows processing.

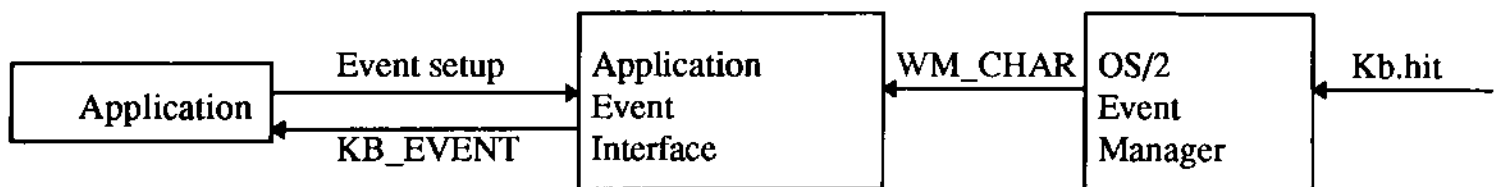


Figure 2.13: OS/2 processing.

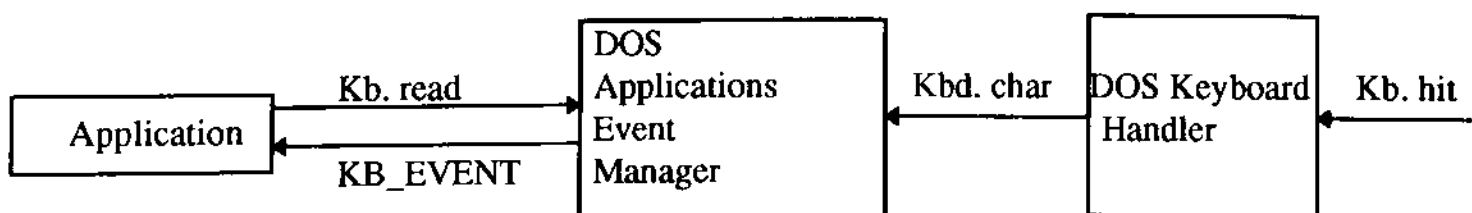


Figure 2.14. DOS processing.

The methods used to develop cross platform user interfaces is to use a compatible shell on a platform and develop the application targeted for that shell or front end software platform.

Developing a native application that is compatible with multiple environments requires the use of development tools and software libraries that provide the application with a consistent programming interface to the environment's control of the console, file system and peripherals. The development tools and libraries isolate the application from the specifics in the platform. For example consider a keyboard event shown in the diagram. The interface library as mentioned before receives the keyboard event for every specific platform. In case of Windows and OS/2, there already exists an operating environment that receives the keyboard press to a WM_CHAR message. The proposed library traps this WM_CHAR message to convert it to its own KB_EVENT message and the application code can process it. In case of DOS platform, there is no operating environment (GUI specific) to trap the incoming hit message. In such a case a DOS event message manager is built by the library to capture the message from the operating system. The native application receives the KB_EVENT as usual from these DOS application messages.

The above example illustrates that it is possible to have a library on any platform to capture a native event keyboard press and uniformly return a KB_EVENT to the application in each case. If one is successful in mapping a set of all the native events possible to a predefined set of events on each

platform, then it is possible to capture all the events transparently across all platforms.

The same applies to even generating user interfaces on all platforms. If one is successful in generating a set of widgets uniformly across all platforms using an abstract interface, and all the events associated with these are mapped to a set of events that are common to all platforms, we then have a cross platform library for each platform that generates user interfaces and events uniformly across all platforms. Thus by writing code once on one platform and porting the code to each platform and compiling it with the platform specific library, we can generate user interfaces on all platforms with one set of source code. This gives us a general idea of how a cross-platform library has to be built and its working.

2.9 State of art in user interfaces

Most of the work in user interface tools can be characterized into three areas [22]:

1. User interface tool kits,
2. User interface management systems,
3. User interface techniques and algorithms.

User interface toolkits are characterized by such systems as Macintosh toolbox, Motif, Openlook and Microsoft Windows. These systems provide a set of reusable routines that can abstract out the low level work needed by common abstraction techniques. Such systems do not attempt to describe or support the user interface as whole. A user interface management system

(UIMS) is characterized by an external description of an interactive dialog. It is intended that the description is created by programmers and can be separated from application code which implements the dialog.

2.10 Research issues in user interfaces

2.10.1 Vision of end user environment

In this section, we will argue for an end-user computing environment that is malleable, one in which the form and content of computing tools are under the control of users. The idea is to give the users the power to construct interfaces and address their individual needs.

Malleable interfaces would offer two distinct advantages.

- a. Let the people know about their problems be in charge of building the tools they need.
- b. In cases where co-development by users and programmers is necessary, malleable interfaces would support a better vision of labor between two groups.

The vision of malleable interfaces offer two kinds of interface development support to users. The first of these is structural allowing them to build new applications by combining existing applications or application components.

The second is behavioral which goes beyond simple composition of components and extends to defining new actions or behaviors in interactive environment.

Structural malleability is characterized by the user reorganizing or reshaping their interfaces.

The user can go beyond simple composition of predefined application components and define new behavior for existing applications.

Current research issues in user interface software is showing promising results in the automatic generation of user interfaces. These tools fall along a continuum where at one end, interface is generated from source code of application without any specification of what the user interface should look like. On other side, the user interface is generated entirely from a demonstration of what it should look and act like without any specification of code.

CHAPTER 3

EXISTING INDUSTRY STANDARDS IN CROSS PLATFORM USER INTERFACES/ PLATFORMS

3.1 Microsoft Foundation Classes (MFC) and Visual C++

The Microsoft foundation Class Library is a robust C++ application framework designed for writing applications in C++ for Microsoft Windows Operating Systems. MFC has been designed to be portable to a number of platforms, allowing applications written on a MFC version to target a variety of different platforms. MFC is also portable to a variety of compiler implementations. Microsoft plans to keep extending and evolving MFC to support new functionality in the operating system. Writing a program on top of MFC makes the program portable to a number of platforms. Since MFC is built on top of Win32 API, any platform that supports the Win32 API can be targeted by a MFC application. The most powerful aspect of MFC portability is that Win32s based platforms are not limited to Microsoft Windows. Through the use of Windows portability layer, MFC applications built on Win32 can also be built to run under the Macintosh System 7 operating system. MFC 3.1 (the version of MFC) applications can target a large number of platforms today. They are:

- Microsoft Windows NT running on Intel processors,
- Microsoft Windows NT running on MIPS processors,
- Microsoft Windows NT running on Alpha AXP,
- Microsoft Windows 3.1 using Win32s,

- Microsoft Windows 95,
- Apple Macintosh (using the 68000 series instruction set).

Efforts are being made to support the PowerPC (running System 7 and Windows NT) in addition to Microsoft Windows API on UNIX platform to make sure developers can port their applications to these applications [18].

3.2 SUIT (Simple User Interface Toolkit)

The Simple User Interface Toolkit (SUIT) [23] developed by the Department of Computer Science at the University of Virginia is probably the only work done on portable user interface toolkits for cross platform user interface development academically.

SUIT is a C subroutine library. SUIT applications run transparently across Macintosh, DOS, UNIX/X, and Silicon Graphics platforms. The architecture that SUIT incorporates in its development will be dealt with later.

3.3 Other Products

There are a lot more other products in the industry that support cross platform user interface development. A few are worth mentioning in this context.

3.3.1 MAINWin

Developed by the Mainsoft Corporation provides an universal MS Windows API for all platforms. That means that one could program on UNIX/X environment with Windows API [17]. It follows a layered architecture (Figure 3.1).

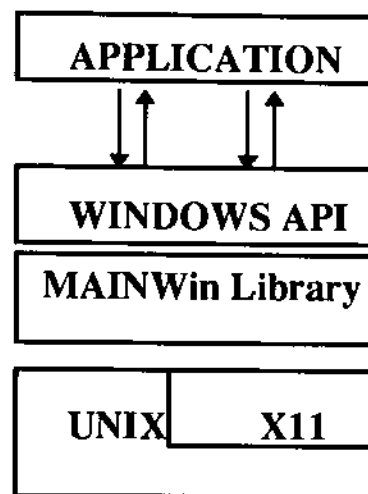


Figure 3.1. MainWin architecture.

3.3.2 zApp

Developed by Inmark corporation, zApp is probably one of the best cross platform development tools available. The package comes with a visual interface builder. Once the interface is generated, the source code is recompiled with the version of zApp for a target platform to provide the target user interface. zApp targets Windows, WindowsNT, DOS Text, DOS Graphics, OS/2 and X/Motif [13] (Figure 3.2).

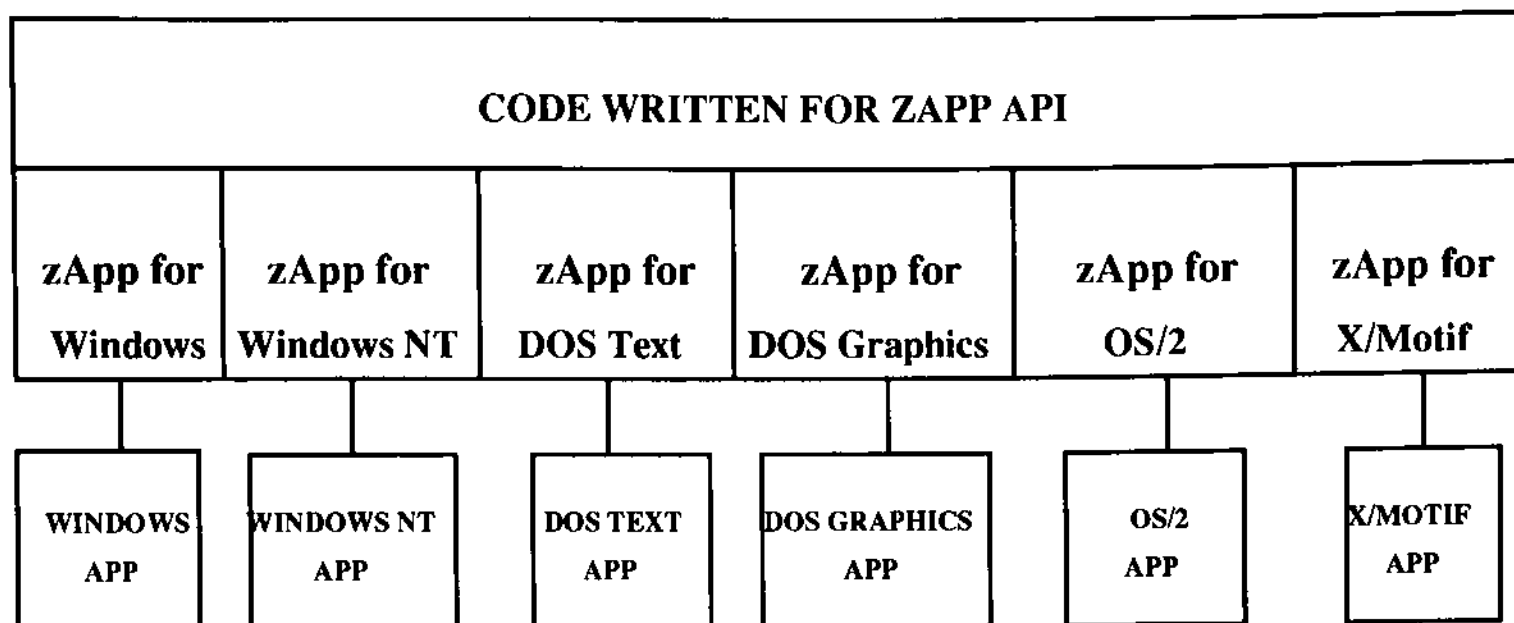


Figure 3.2. zApp architecture.

3.3.3 WIND/U

Wind/U family of products from Bristol Technologies allows programmers to use Visual C++ and Microsoft Windows API to develop applications on UNIX/Motif applications with context sensitive help, Postscript and PCL printing. It works similar to MAINWin.

A few other products in the industry that are worth mentioning are Zinc application framework, ILOG tools and libraries, WM-Motif and Open-UI [3].

CHAPTER 4

EXISTING TECHNIQUES AND METHODOLOGY

4.1 Current Existing Techniques

Even though there are many toolkits¹ on different platforms, there are several similarities existing between them. For example, all toolkits have some sort of menu, listbox, button, radiobutton and so on. The only challenge existing is to write user interface software with the minimal effort and hence target it to different platforms with a single source code.

This challenge has been well taken by the industry hence leading to a number of libraries and packages trying to hide the underlying differences among different platforms.

There are two techniques that are widely followed when developing cross platform user interface software. They are the method of layered software and the method of virtual widgets.

¹ A toolkit is a library of widgets that can be called by an application program.

4.1.1 Layered Software

The figure below (Figure 4.1) shows the layered SUIT architecture.

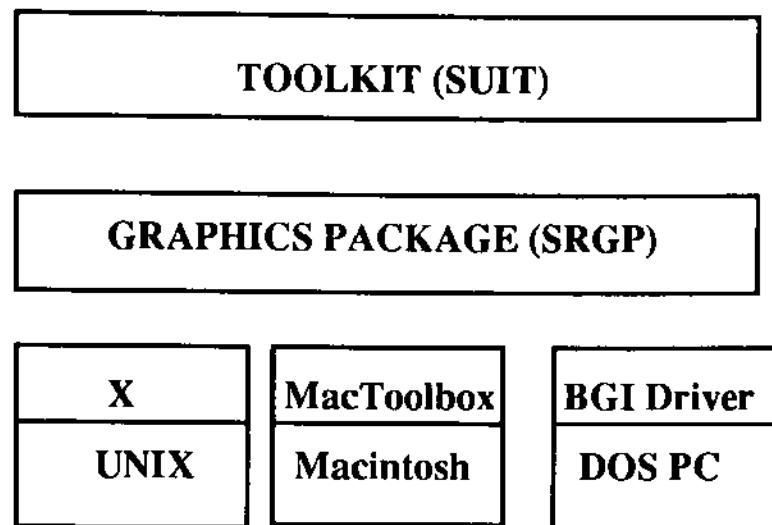


Figure 4.1. SUIT architecture.

The Simple User Interface Toolkit mentioned before follows this architecture. The toolkit is implemented in ANSI C. For writing graphics to the screen, the package uses the Simple Raster Graphics Package (SRGP) which is described and distributed by Addison-Wesley publishing company with their second edition of the book *Computer Graphics: Principles and Practice* by Foley, van Dam et al. [34]. SRGP is ported to Turbo C on DOS , X Windows and Macintosh. The toolkit (SUIT) which is built on top of the SRGP runs transparently across all systems. The main disadvantage of this method is that, since the toolkit uses the graphics package, the user interfaces that are developed across different platforms have the same look and feel but applications on a particular platform will not conform to the characteristics of other applications on the platform.

SUIT currently supports Silicon Graphics, HP 9000/HPUX and IBM RS-6000/AIX for Motif, SunOS for Motif, SunOS for Openlook, Macintosh, MS Windows, DOS Graphics and DOS Text.

The SGP (Simple Graphics Package) is a device independent subroutine package that provides program portability. Because SGP makes the application program independent of specific hardware, it can be used with different graphic configurations with minimal changes.

4.1.2 Virtual Toolkits

Virtual toolkits work in a way to hide the differences among various toolkits by providing virtual widgets, which in turn are mapped onto the widgets in each toolkit. The programmer writes code once using the virtual toolkit and the code runs without any changes on different platforms. "The goal of supporting the creation of platform independent programs is to allow the total effort to be less than the sum of the efforts needed on the individual platforms"[5, p.29]. The final goal being that there should be a similar uncompromised performance on all platforms.

For example, the virtual toolkit XVT provides a C/C++ interface that links to the actual Motif, OpenLook, Macintosh, MS-Windows and the OS/2 Presentation Manager toolkits and hides their differences. It is to be noticed here that the job of the toolkit serves only to provide a different interface to the widget already existing on the platform. The programmer is confined to a smaller library of routines, that is common to all the platforms. This serves

only in creating the user interface skeleton necessary. The programmer has to have an in-depth knowledge of the graphics primitives followed on that platform to create the necessary application.

Another way of implementing virtual toolkits is to reimplement the widgets on a particular platform. The toolkit may incorporate a complex graphics package to do this. Also, the programmer need not know the graphics primitives on a platform since the toolkit uses a graphics package to generate the user interface. This method of developing a virtual toolkit is nothing but the method described before for the SUIT. Since the applications that are developed using this type of toolkit does not conform to the look and feel of other applications on a platform, they can hardly be considered as virtual toolkits. Galaxy by Visix Software Inc. follows this method of generating user interfaces.

The main advantages of applications built using the layered software technique is that a single application built across different platforms have the same look and feel and programmers need not have any knowledge of the graphics primitives followed on that platform. The disadvantage is that the applications might run slower at times. The advantages of applications built using virtual toolkits is that the applications have the same look and feel as other applications on the platform. The disadvantage is that the programmer has to have an in-depth knowledge of the graphic primitives followed on that platform [21].

CHAPTER 5

CROSS PLATFORM USER INTERFACES

Chapter 4 explained the techniques used for building cross platform user interfaces. The research further delves into how these techniques can be supported by advanced software architectural principles. In doing so, some of the design patterns that can be incorporated are explained in this chapter. The designer of a cross platform user interface will be reusing these design patterns a number of times if he were to program in an object oriented language. Also, the different models used to describe software architecture of user interfaces are described in Chapter 6. Models are used to evaluate any architecture in terms of desired goals. As far as cross platform user interfaces are concerned, the goals for design would be to buffer the user interface for changes in the computing platform and the underlying windowing system, the ultimate goal of which would be to generate uniform user interfaces on all platforms using one set of source code. A few case studies of user interface architectures are examined in the coming chapter and the user interfaces are examined for cross platform portability using one of the models to describe user interface architectures. The requirements for cross platform portability are examined in each case.

The book on design patterns by Erich Gamma and others [10] discusses 23 design patterns that are used in developing reusable object oriented software. In order to better understand the implementation details of

cross platform user interfaces, it is worthwhile having a look at two of the patterns discussed in the book.

5.1 Abstract Factory

Consider the user interface toolkit that supports multiple look and feel such as Motif and Presentation Manager. Different look and feel standards define different appearances and behaviors for user interface "widgets" like scrollbars, windows and buttons. To be portable across look and feel standards, an application should not hard code its widgets for a particular look and feel.

The problem is solved by defining an Abstract WidgetFactory class that declares an interface for creating each basic kind of widget (Figure 5.1). There is also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look and feel standards. For example, the CreateScrollBar operation on the MotifWidgetFactory instantiates and returns a Motif scroll bar, while the corresponding operation on the PMWidgetfactory returns a scrollbar for Presentation Manager. Clients create widgets solely through the WidgetFactory interface and have no knowledge of the classes that implement widgets for a particular look and feel.

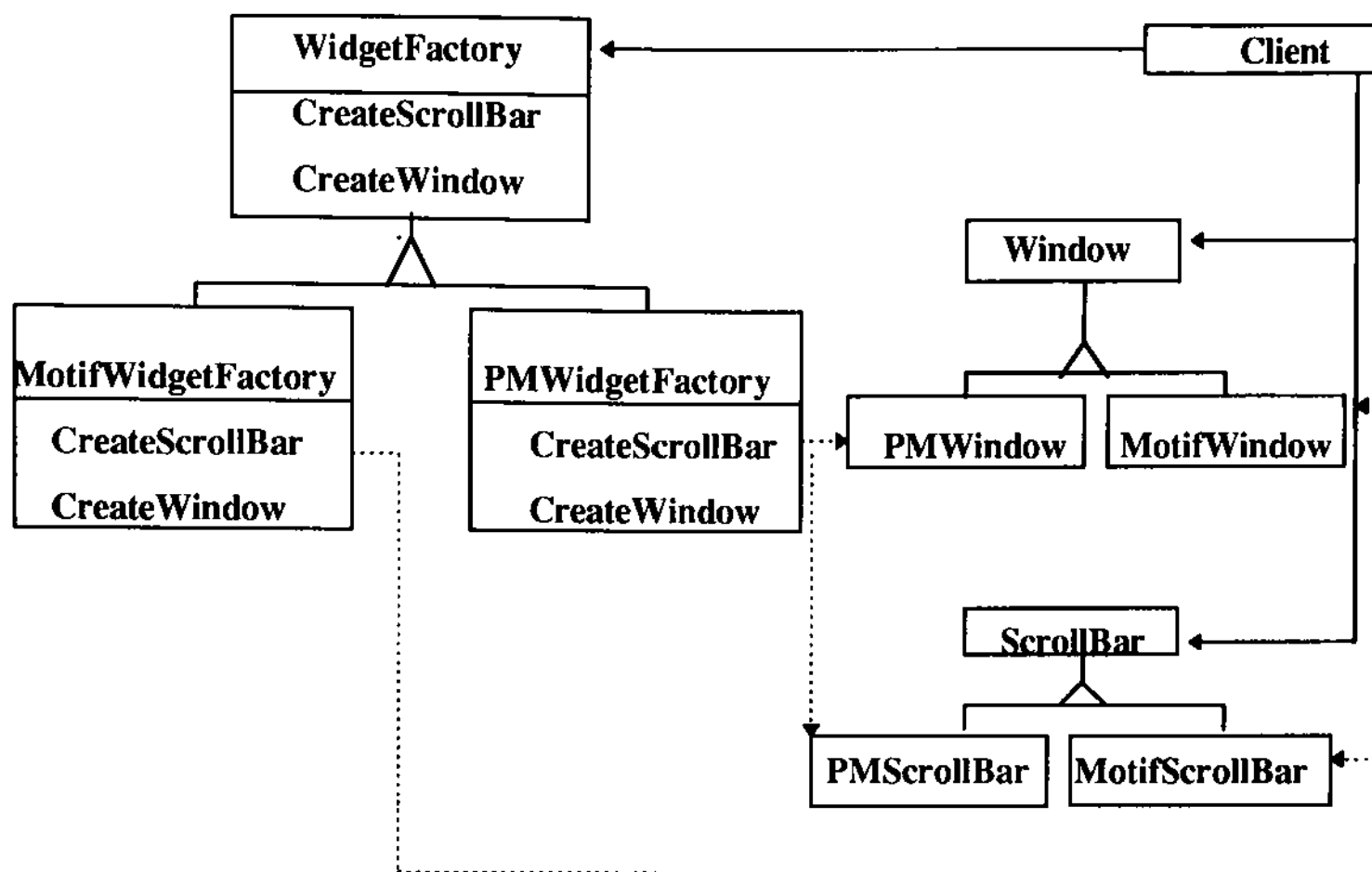


Figure 5.1. Abstract factory.

Abstract Factory pattern is used when any of the following conditions have to be satisfied:

1. A system should be independent of how its products are created, composed and represented.
2. A system should be configured with one of multiple families of products.
3. A family of related product objects is designed to be used together, and you need to enforce this constraint.
4. One wants to provide a class library of products, and wants to reveal just their user interfaces, and not their implementations.

5.2 Bridge

When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance. An abstract class defines the interface to an abstraction, and concrete subclasses implement it in different ways but this approach is not flexible. Inheritance binds an implementation to the abstraction permanently which makes it difficult to modify, extend and reuse abstractions independently.

Consider the implementation of a portable window abstraction in a user interface toolkit (Figure 5.2). This abstraction would enable us to write applications that works both on the X Window system and IBM's Presentation Manager. This approach has two drawbacks.

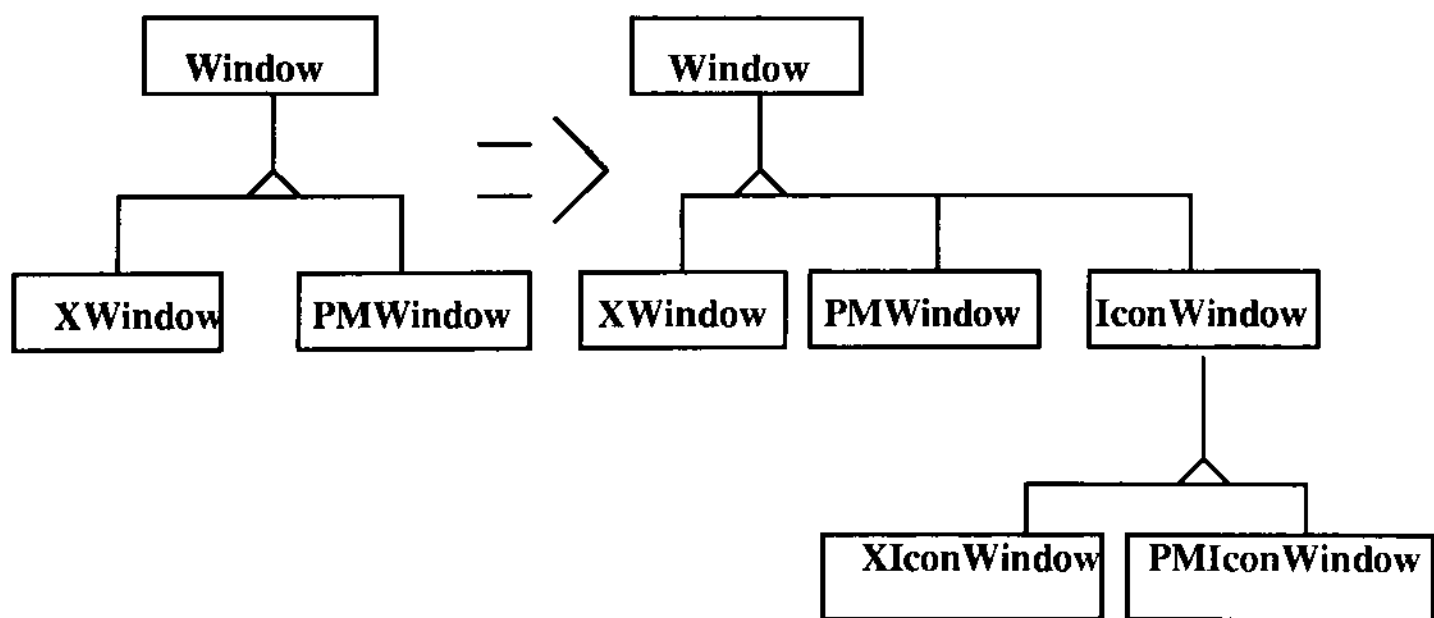


Figure 5.2. Portable window abstraction.

1. It is inconvenient to extend window abstraction to different kinds of windows or new platforms. For instance, to support icon windows on

both platforms, we need to implement two new classes XIconWindow and PMIconWindow.

2. It makes client code platform dependent. Creating a PMIconWindow binds the IconWindow abstraction to PMIconWindow implementation. This in turn makes it harder to port the client code to other platforms.

The main principle is that the client should be able to write code which is not platform specific which means that the client's code must be platform independent. The bridge pattern addresses these problems by separating the abstraction hierarchy from its implementation hierarchy as shown in Figure 5.3.

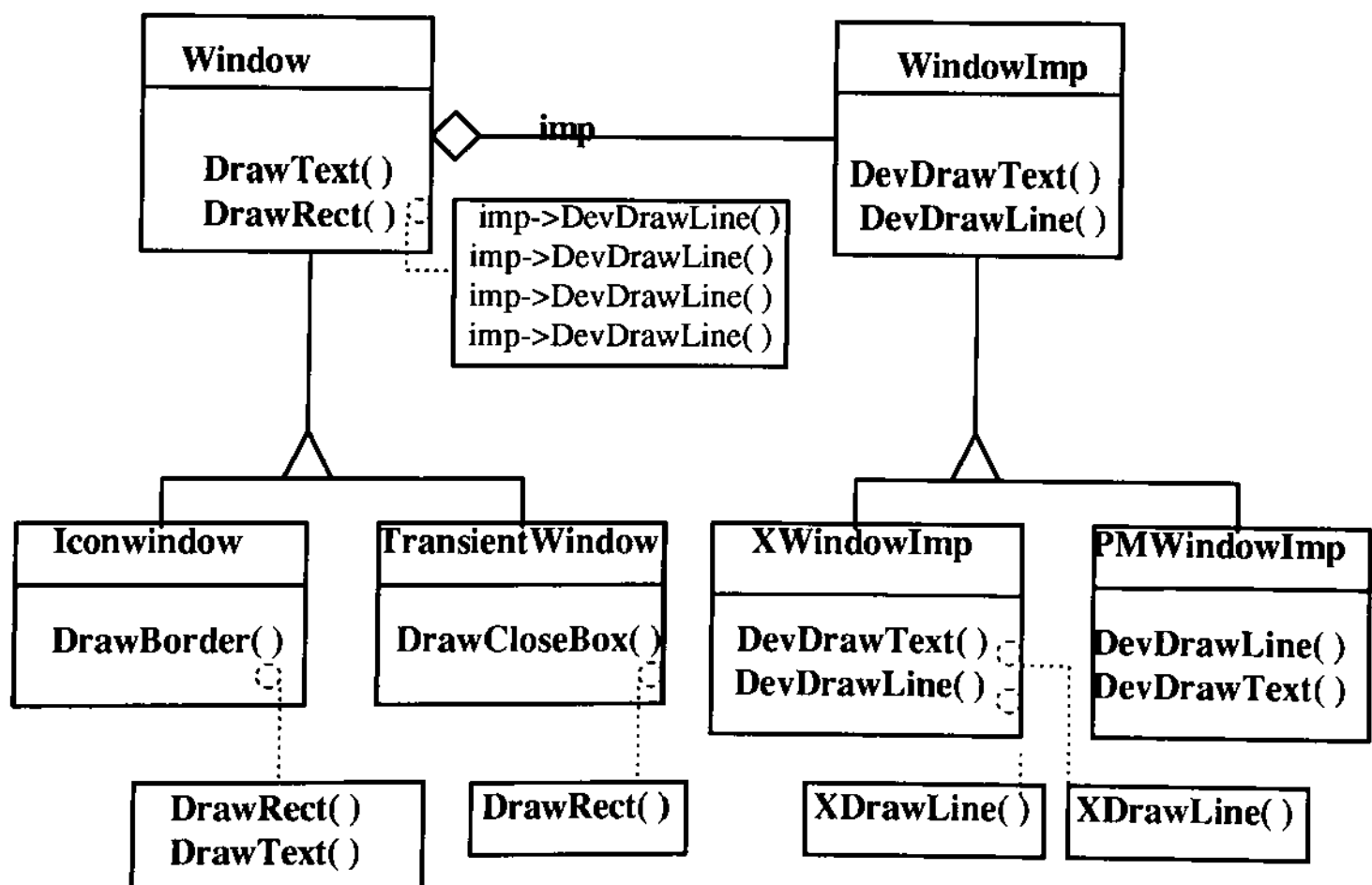


Figure 5.3. Bridge pattern.

The Bridge pattern is used when any of the following conditions have to be satisfied:

1. One wants to avoid permanent binding between an abstraction and implementation.
2. Both abstractions and implementations should be extensible by subclassing.
3. Changes in implementation should not affect client code.

It should be possible to switch the implementations between any two different types for a given platform at a particular time. MAINWin supports both the Windows and Motif looks and feels for its user interfaces on UNIX workstations. The end user can change looks on the fly at runtime. Using a bridge pattern, support for multiple looks and feels on a single platform should be possible since the abstraction hierarchy is passed a pointer to the implementation at a particular time.

These two patterns describe implementations that can be incorporated when developing cross platform user interfaces with C++. If one wants to develop user interfaces with C, a different pattern of programming style has to be followed so as to generate user interfaces with one set of source code on all platforms. The following chapter describes some of the architectures that are used for developing cross platform user interfaces in C and C++.

CHAPTER 6

SOFTWARE ARCHITECTURE FOR CROSS PLATFORM USER INTERFACES

6.1 Introduction

The idea of a metamodel is introduced with two purposes: to derive any number of architectures, depending upon a particular developer's goals, and to evaluate any proposed architecture in terms of desired goals.

Design goals can be expressed in terms of criteria for evaluating either the end product or development process. Some of the common criteria to be considered when designing an interactive system are [2]:

- Target system performance
 - size,
 - speed;
- Productivity of system development tools;
- Buffering from changes in
 - application domain,
 - medium,
 - hardware platform,
 - interaction toolkit;
- Conceptual simplicity;
- Reuse of code;
- Meeting functional requirements;
- Quality of resulting user interfaces;

- Cost of development;
- Complexity of specification;
- Time for target system to reach its end user;
- Developer's inabilities;
- Target system extensibility;
- Adherence to standards;
- Compatibility with other systems;
- Complexity of application code;
- Complexity of dialog requirements.

Regardless of the criteria selected as most important, all designs must provide a certain set of functionalities. There will be data representations and functions necessary in an interactive system. Different systems place different degrees of emphasis on each function, but every interactive system must perform these functions in some manner.

As mentioned previously, the Arch/Slinky metamodel of user interface architectures identified Functional Core (FC), Functional Core Adapter (FCA), Logical Interaction (LI) and Physical Interaction (PI). The user interface toolkits explained in this chapter will be evaluated for buffering for changes in the operating platform. The objective will be to keep the Logical Interaction (LI) component separate from the Physical Interaction (PI).

6.2 Models for describing user interface software architecture

6.2.1 The Arch Model

The figure below (Figure 6.1) depicts the Arch model.

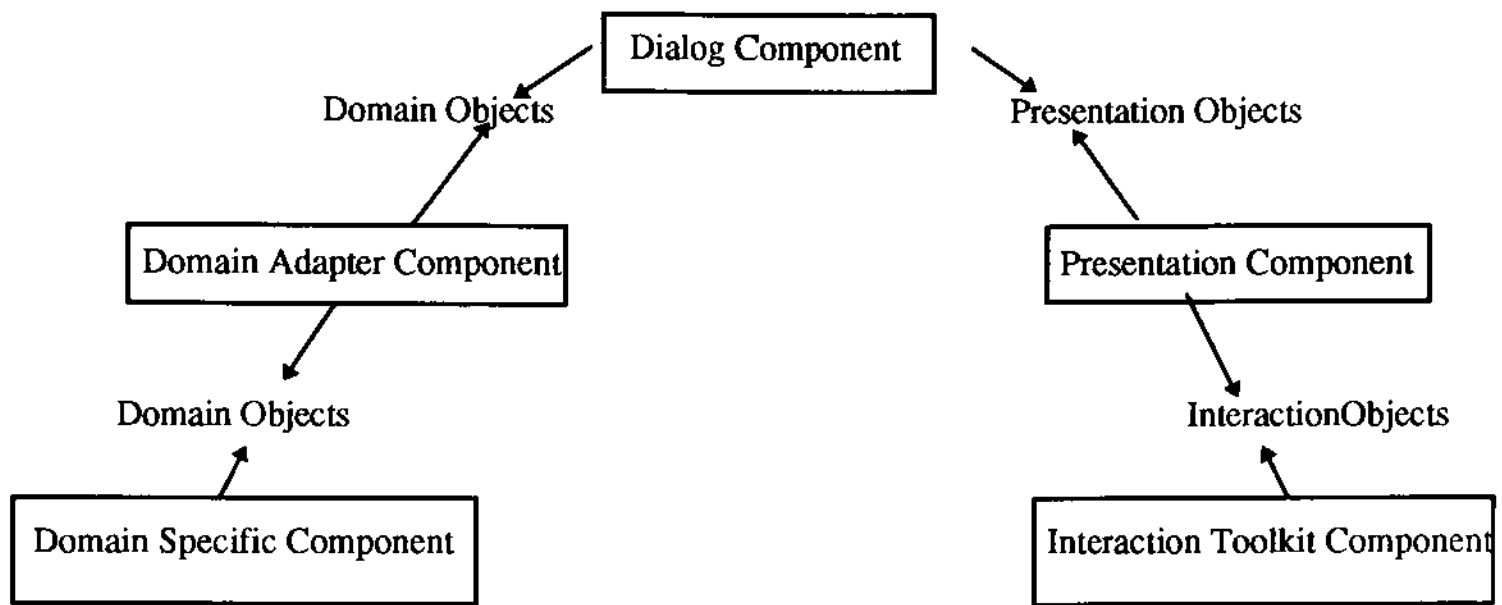


Figure 6.1. Arch model.

There are five components in this model. They are as listed below:

- *Domain Specific Component* controls, manipulates and retrieves domain data and performs other domain related functions.
- *Interaction Toolkit Component* implements the physical interaction with the end user via hardware and software.
- *Dialog Component* has responsibility for task level sequences, both for user and for the portion of application domain sequencing.
- *Presentation Component* is a mediation, or a buffer component between the Dialogue and Interaction Toolkit components that provides a set of toolkit independent objects for use by the dialogue design.
- *Domain-Adapter Component* is a mediation component between the Dialogue and Domain-Specific components. Domain related tasks

required for human operation of the system but not available in Domain Specific component are implemented here.

- *Domain Objects* are employed by both the Domain Specific and Domain Adapter components, but instances of these objects are created by two components for different purposes.
- *Presentation Objects* are virtual interaction objects that control user interactions. Presentation objects include data to be presented to the user and events to be generated by the user.
- *Interaction Objects* are specially designed instances of media specific methods for interacting with the user. Interaction Objects are supplied by the Interaction Toolkit and may be primitive or complex.

6.2.2 The Slinky Metamodel

The Slinky Metamodel is a more general description of the Arch Model. We can derive a set of Arch models from the Slinky Metamodel. The Arch model serves to provide a stable development environment where goals have been set and choices have been made. The term “Slinky” was selected to emphasize that functionalities can shift from component to component in an architecture depending upon the goals of the developers, their weighting of development criteria and type of system to be implemented.

To clarify the concept of shifting functionalities, consider an example where a function in a Domain Specific Component was later implemented in an Interaction Toolkit.

6.2.3 PAC Model

The PAC model structures an interactive application into three parts: Presentation, Abstraction and Control.

- The Presentation defines the concrete syntax of the application, i.e., the input and output behavior of the application as perceived by the user [4].
- The Abstraction part corresponds to the semantics of the application. It implements the functions that the application is able to perform. These functions are as a result of task analysis.
- The Control part maintains the consistency between the abstract entities that were involved in the interaction and implemented in the Abstraction part and their presentation to the user. It is intended to hold the context of the overall interaction between the user and the application.

Consider Figure 6.2. The presentation of an application is a set of entities, called interactive objects, specialized in man-machine communication.

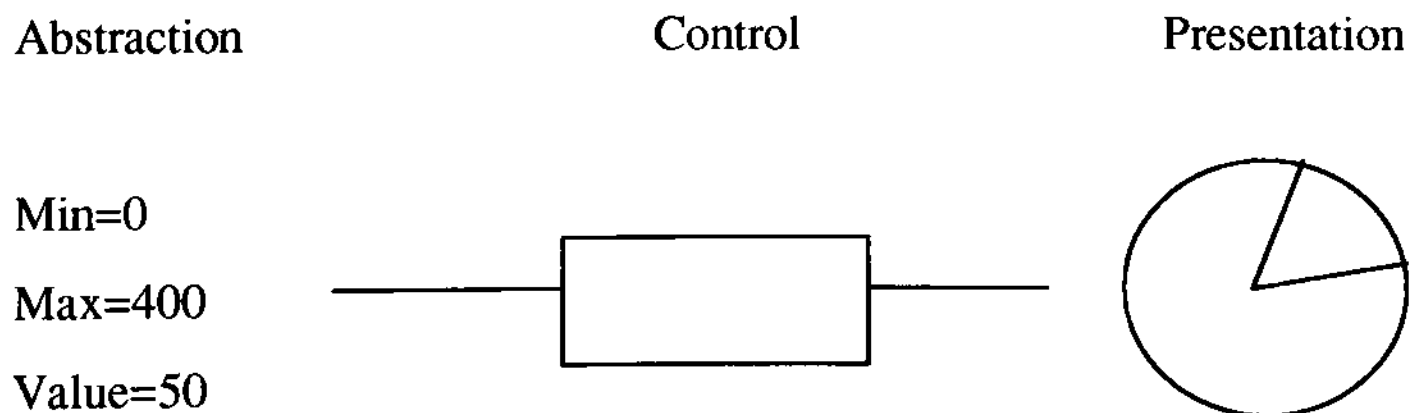


Figure 6.2. An elementary interactive object.

The presentation consists of:

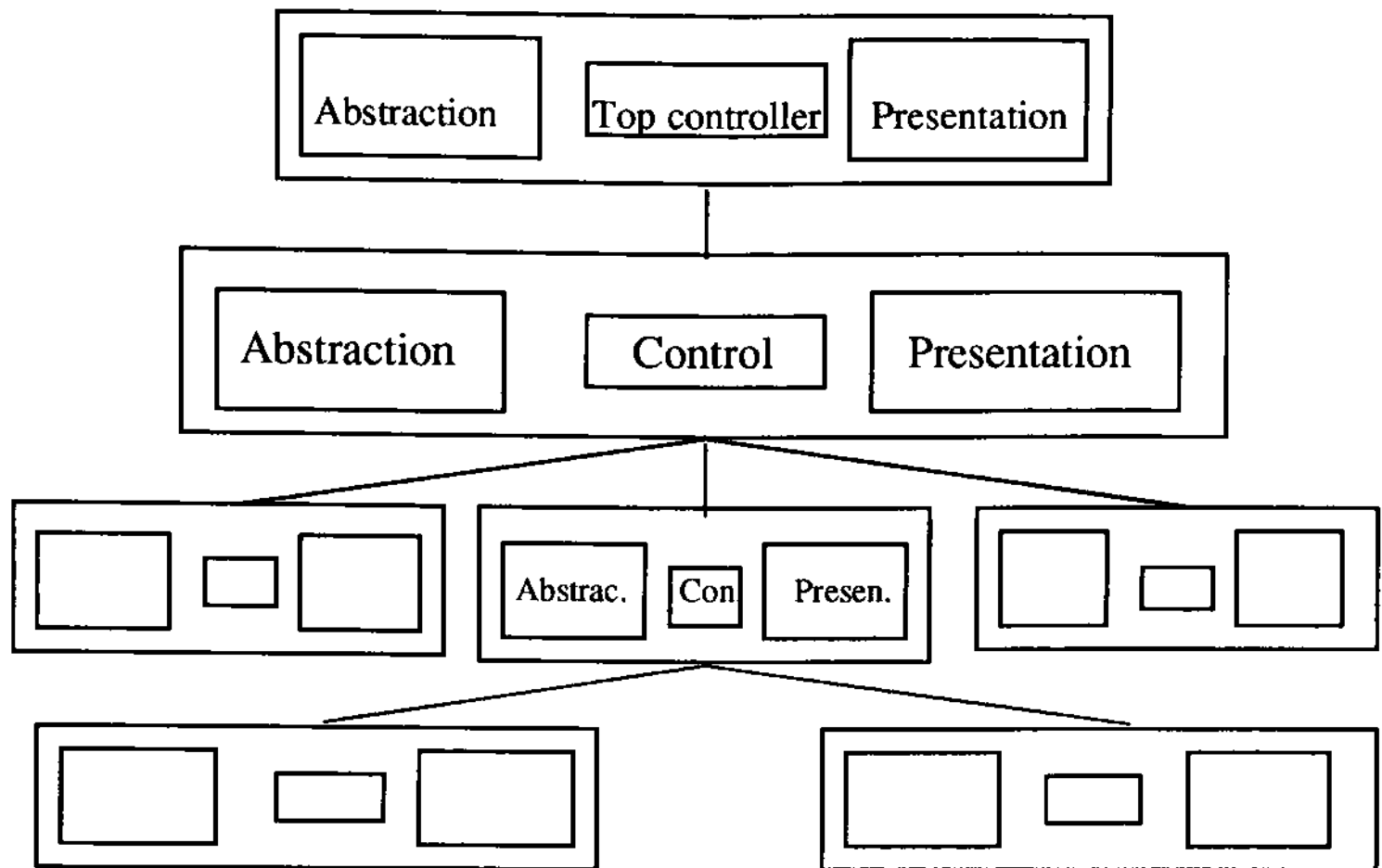
- An output, circular shape or color for each piece of the pie;
- An input mode for performing mouse actions to interactively change the relative size of pieces.

The abstraction comprises of an integer value within the range of two integer limits.

Also, the Control maintains the consistency between the Presentation and Abstraction. If the user changes the size of the pie, the Control reflects the change in the abstraction entities (integer value in this case). The converse or reverse of this is also true where the Presentation updates the figure in response to any change in the value of the integer (abstraction entity).

In an application, if PAC is recursively applied at every level of abstraction of the user interface, everything in an interactive application is a PAC object. In such a case, the top controller will hold the main control loop of the dialogue but hooks should be provided for mixed control.

A typical PAC structure of an interactive application is shown in Figure 6.3.



Abstrac.: Abstraction, Con.: Control , Presen.: Presentation

Figure 6.3. PAC structure.

6.3 Case studies of Software Architecture

A few architectures of user interfaces are presented here that are analyzed in accordance with Arch Slinky model for user interfaces.

6.3.1 Serpent

An UIMS (User Interface Management System) which is nothing but a set of tools for the specification and execution of the user interface portion of the system. A UIMS provides tools for the specification of the static layout portion of the user interface, for the specification of the dynamic portion, and for the execution of the specifications. A UIMS also separates the user

interface portion of a system from the functional portion, allowing for modifications to the user interface with minimal impact on the remainder of the system.

The Serpent is a UIMS that supports the incremental development of the user interface from prototyping through production and maintenance. The Serpent supports the X Window System, Version 11, with the Athena Widget set and the OSF Motif Widget Set [33].

Figure 6.4 below shows all the five basic functional roles of user interface architecture, as identified before, mapped onto Serpent's software architecture.

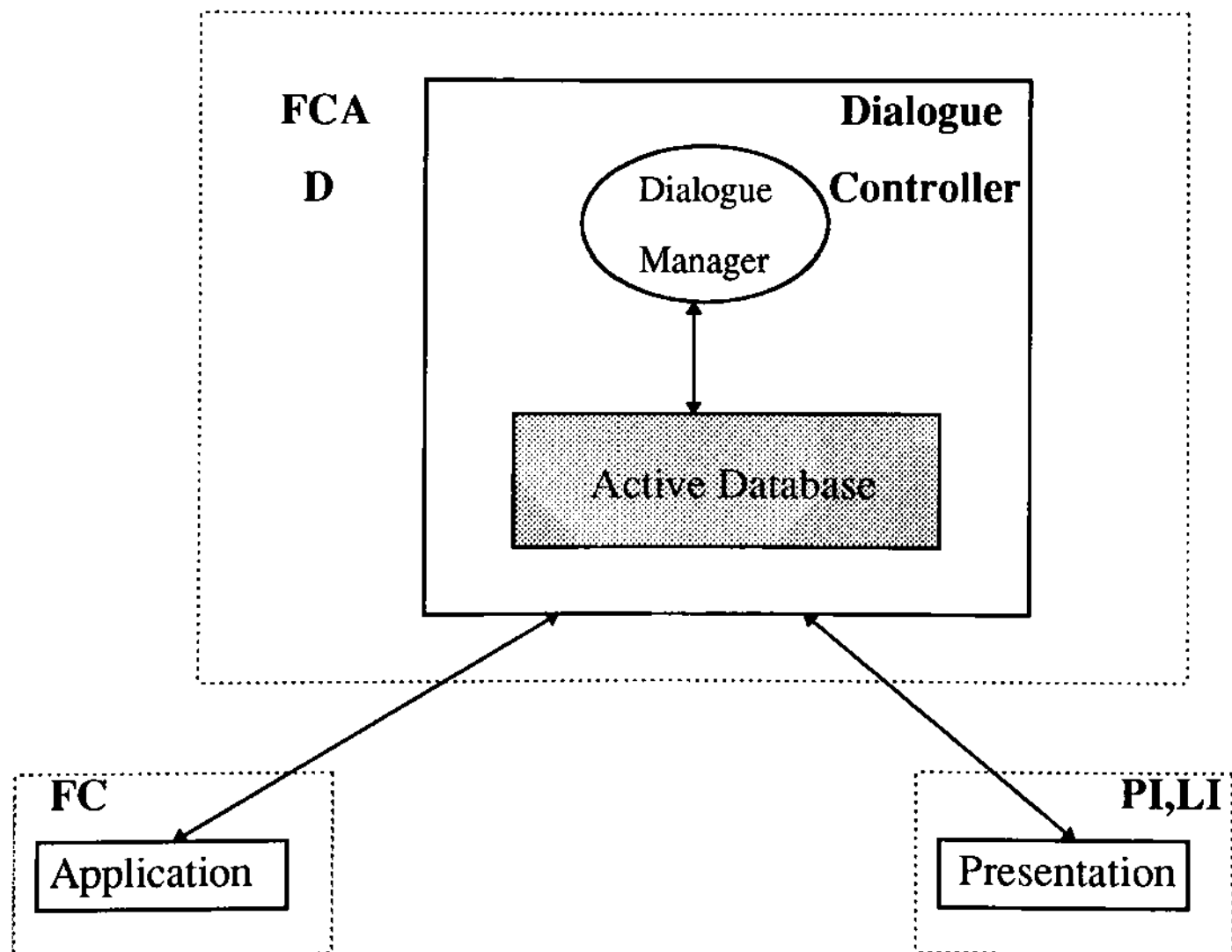


Figure 6.4. Serpent architecture.

Application modules contain the computational semantics required for some domain application. Although there can theoretically be many different applications contained within a given runtime instance of Serpent, there is typically only one application. Given that the application and presentation modules are separate, there must be a way to coordinate a given application component with a presentation component. This is the purpose of dialogue controller. The dialogue component mediates the user's interaction with an application, through the control of the presentation.

All the communication between Serpent components is mediated by constraints on shared data in the database. This structure is implemented as an active database. When the values in the database change, they are automatically communicated to any component which is registered as being interested in the data. This global database physically resides in the same process as the dialogue controller but is logically independent from all Serpent components.

Several points of interest are to be noted here:

- There is no structural separation between the PI and LI roles in the presentation component.
- There is no structural separation between the FCA and D components in the dialogue controller component.
- All of Serpent's components are monolithic, that is, they provide no architectural support for subdivision of functionality within a component [15].

6.3.1.1 Changing the Physical Interaction

As far as changing the physical interaction in Serpent is concerned, we have to have an application running under one toolkit and we want to change to another which is supported by the Serpent run-time environment. The LI and PI components are combined into a single run-time entity. The architecture does not separate the PI component from the rest of the system. If the dialogue were to be written utilizing a sufficiently abstract set of virtual objects, then changing the physical interaction would be easier. Since this is not done, changing the physical interaction is difficult here.

6.3.2 Chiron

Chiron is a user interface system which is built with a client server style. It currently supports X Windows. The Chiron was built with the expressed goals of addressing life cycle concerns of maintainability and sensitivity to environmental changes. The architecture consists of a client and server. The client consists of an application, which exports a number of abstract data types (ADTs) which Chiron encapsulates with Dispatchers. Dispatchers communicate with Artists, which maintain abstract representations of their associated ADTs in terms of an Abstract Depiction Library (ADL).

The Chiron server consists of the ADL; a virtual window system, which translates the abstract interface descriptions into concrete ones; and an instruction/event interpreter (Figure 6.5). The instruction/event interpreter responds to requests from Artists to change the abstract description and

translates those requests into changes to the presentation. It also responds to events from users and translates those back into Artist requests.

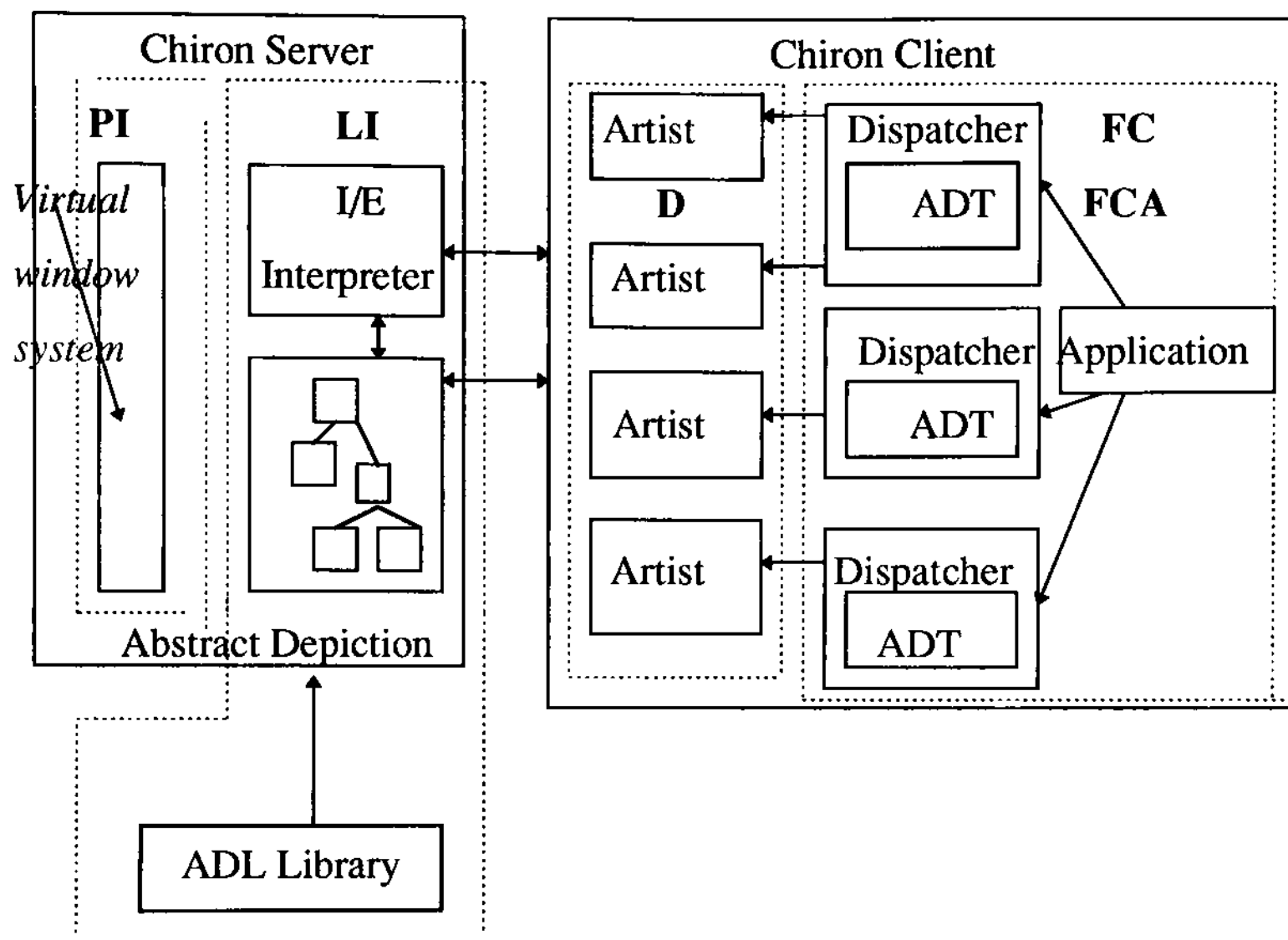


Figure 6.5. Chiron architecture with functional roles annotated.

The Chiron architecture clearly separates the application (functional core) from the rest of the system which was built with the expressed goal of minimizing sensitivity to environmental changes. The functional core adapter could live in the ADTs, or in the Artists. It seems clear that the Artists contain some of the dialogue, for example, maintaining a correspondence between objects from the application domain and interface objects, from the presentation domain.

The physical interaction component appears to be in Chiron's Virtual Window System component, and the logical interaction component is encapsulated within the ADL. As a result of this characterization, we can provisionally annotate the Chiron Architecture as shown in Figure 6.5. This re-characterization indicates the logical division of functionality in Chiron, according to the Arch/Slinky metamodel.

6.3.2.1 Changing the Physical Interaction

Chiron goes one step beyond Serpent in providing architectural support for changing the PI component. Chiron has isolated the PI component and the LI component in separate structures within the Chiron server. This architectural isolation means that the LI and PI components are logically and physically separate, and must communicate via a well defined interface. The PI component can be replaced or inserted into the architecture as long as they comply with the interface and LI conventions.

6.3.3 Garnet

The emphasis of Garnet's architecture is on control of the runtime behavior of interaction objects and the visual aspects of the interface. Garnet's architecture consists of a single process with Common Lisp and the X11 window system as its functional foundations. On top of this base, the Garnet toolkit has been constructed, and on top of the toolkit are high level Garnet tools. Figure 6.6 shows the layered system of Garnet architecture.

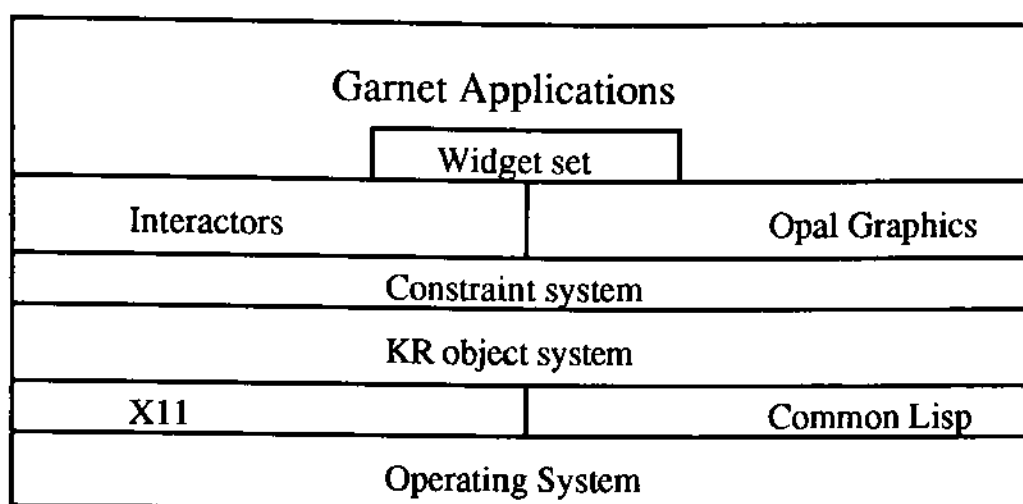


Figure 6.6. Garnet architecture.

However, Garnet is not a strictly layered system. In a strictly layered system, the n th layer hides all details of the layers $n-1$ and lower, and provides services to layer $n+1$. In Garnet, Common Lisp, the KR object system, and the constraint system are used directly by all layers. The Interactors and Opal Graphics together strictly hide all the X11 calls. Thus, the widgets and applications do not have any operating system or window manager calls in them, only calls the interactors, opal graphics, constraints and object system. Figure 6.7 shows the Garnet architecture with functional roles annotated.

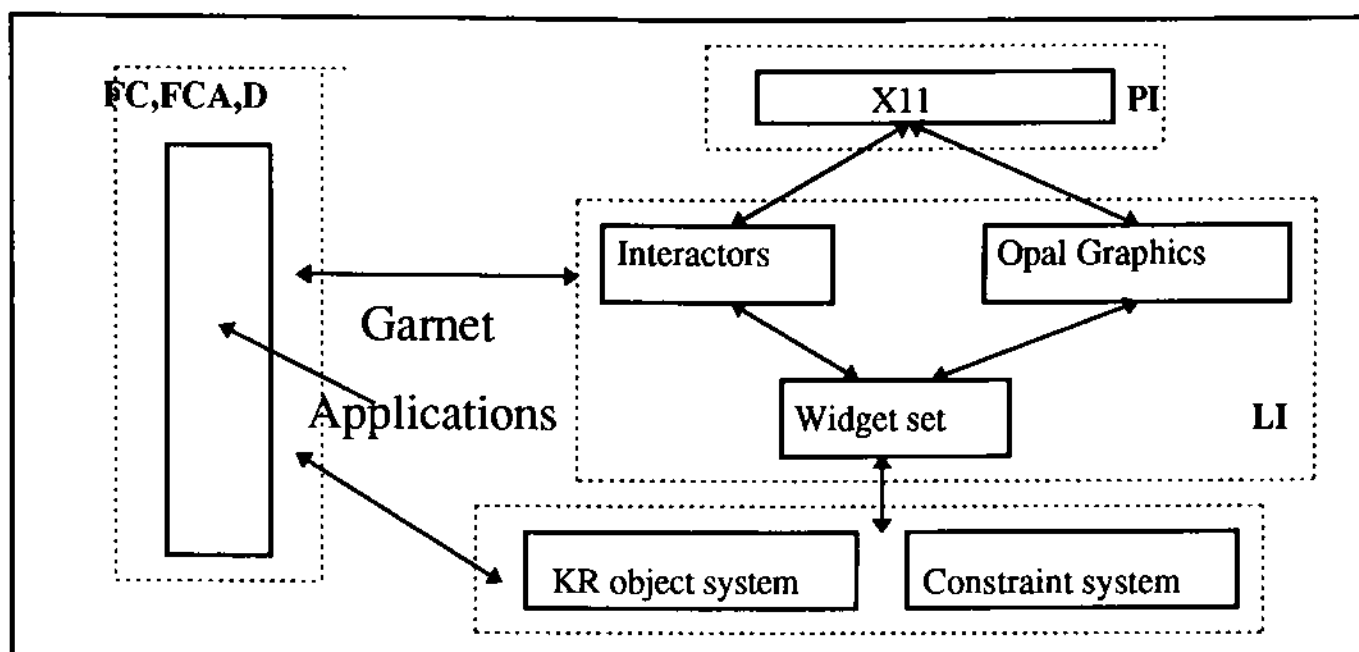


Figure 6.7. Garnet architecture with functional roles annotated.

The X11 window system is the Garnet's physical interaction component. The Widget set and Opal Graphics collectively comprise Garnet's logical interaction (LI) component, and make use of the object and constraint system and services. The arrows in Figure 6.7 have been deliberately drawn as though they connected the layers together, rather than showing separate arrows between each of the elements. This is because Garnet provides the object and constraint systems and logical interaction components as services which are potentially available to all layers, rather than as encapsulated structures. The physical interaction of Garnet is strictly hidden from other layers and it can be accessed through the Widget set, Interactors and Opal Graphics.

Garnet subdivides dialogue functionality into three distinct parts and logical interaction functionality into three parts. Application functionality however is not subdivided, and is not separated in specification or at runtime,

from the rest of the system. The connections between FC/FCA/D component and LI component is shown as single link.

6.3.3.1 Changing the Physical Interaction

Garnet has successfully isolated the PI component to a single component and has hidden any X11 dependencies behind the LI component (Opal Graphics and Interactors). Since Garnet has successfully separated the PI component we can claim that changing this component is supported by the architecture and should be easy. In fact a Macintosh release of Garnet is planned.

It can thus be observed that as far as the PI (Physical Interaction) component can be changed with the LI (Logical Interaction) component being retained, a toolkit can achieve cross platform transfers. It should however be remembered that the toolkit must adhere to an abstract interface pattern across all platforms.

6.3.4 Visual C++ and MFC

Visual C++ is a new way to program for Windows. For someone who has programmed using the Software Development Environment at there is too much of complexity and there is a WndProc() routine by Windows. To alleviate this Visual C++ class library application framework

Interlibrary Loan

742 - 2239

from the rest of the system. The connections between FC/FCA/D component and LI component is shown as single link.

6.3.3.1 Changing the Physical Interaction

Garnet has successfully isolated the PI component to a single component and has hidden any X11 dependencies behind the LI component (Opal Graphics and Interactors). Since Garnet has successfully separated the PI component we can claim that changing this component is supported by the architecture and should be easy. In fact a Macintosh release of Garnet is planned.

It can thus be observed that as far as the PI (Physical Interaction) component can be changed with the LI (Logical Interaction) component being retained, a toolkit can achieve cross platform transfers. It should however be remembered that the toolkit must adhere to an abstract interface pattern across all platforms.

6.3.4 Visual C++ and MFC

The Microsoft Foundation Class Library is a new way to program for the Microsoft Windows Environments. For someone who has programmed for the Microsoft Windows Environments using the Software Development Kit supplied by Microsoft, it is obvious that there is too much of complexity involved. There is one Winmain() routine and there is a WndProc() routine that traps all the messages returned by Windows. To alleviate this complexity, Microsoft came up with a class library application framework

using which one could program easily in Windows Environment. MFC has become a success because it has managed to ameliorate the major problems associated with programming for Windows without obscuring the basic Windows interface. It is always to the advantage of an MFC programmer to know about the underlying event driven model of Windows.

The library classes mediate between the programmer and Windows API. Though they impose little time and space overhead, they provide a large benefit in readability and reusability.

Five classes derived directly from CObject are CWinApp, CWnd, CMenu, CGdiObject and CDC interface classes.

All the member functions of MFC classes are mapped onto the related functions in Windows API.

Building an MFC application involves three sorts of classes: the programmer's own classes, the supplied MFC classes and the classes derived from MFC classes. There are some important MFC classes to be noted. They are:

1. *CWinApp* is the basis of MFC applications for Windows. All applications for Windows developed with the MFC library derive an application-specific class from CWinApp.
2. *CFrameWnd* implements an ordinary, frame tiled window like the main window of most applications. Almost all MFC applications derive from CFrameWnd.
3. *CMDIFrameWnd* and *CMDIChildWnd* are used instead of CFrameWnd in multiple document interface (MDI) application.

4. *CDialog* and *CModalDialog* implement generic dialog boxes. Applications derive classes from these classes to create more specialized dialog boxes.

A typical application is built around a pair of objects called a document and a view (Figure 6.8). A document is an object that manages the data used by an application. A view is an object that manages the display of that data on the screen. When a user launches an application, its view object is the user's window into the program data. The view object determines how the user sees the data in the program's document object and how the user interacts with the data.

Visual C++ displays views inside windows called document frame windows. In an SDI (Single Document Interface) application which has just one window, the document frame window is also the application's main frame window. In case of SDI, the mainframe window is derived from *CFrameWnd* and in case of MDI (Multiple Document Interface), the mainframe window is derived from *CMDIFrameWnd*.

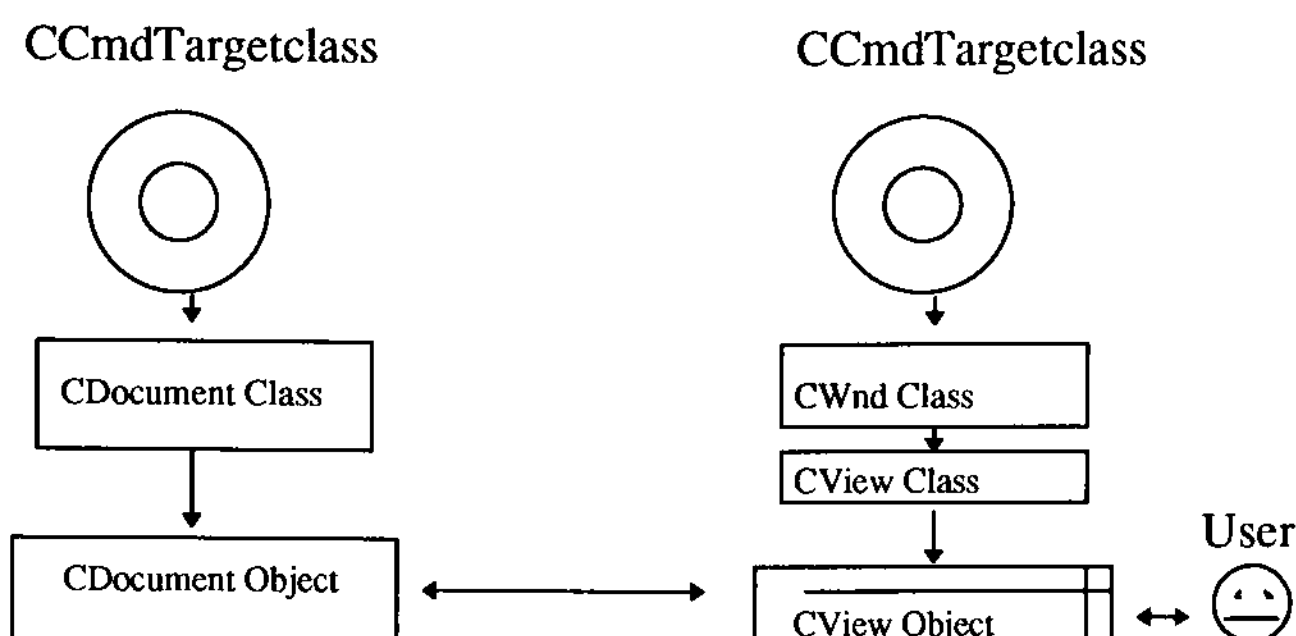


Figure 6.8. CDocument and CView hierarchy.

Figure 6.8 shows how document objects descend from the CCmdTarget and CDocument classes and how view objects descend from CWnd and CView classes. It also shows that although a view object is closely associated with a document object and serves as an interface between the data in a document object and user of the program that manipulates the data.

Visual C++ comes with a Visual Workbench which has a visual tool called AppWizard. When applications are created with AppWizard, it automatically provides the framework of a document and places the code that implements it into a file.

6.3.4.1 Structure of a Visual C++ application

The listing below (Figure 6.9) shows a typical Visual C++ application.

```
// hello.h

#ifndef _HELLO_H
#define _HELLO_H

class CMainWindow: public CFrameWnd
{
public:
    CMainWindow();
    afx_msg void OnPaint();
    afx_msg void OnAbout();
    DECLARE_MESSAGE_MAP();
};

class CTheApp: public CWinApp
{
public:
    BOOL InitInstance();
};

#endif

// hello.cpp

#include <afxwin.h>

#include "resource.h"
```

Figure 6.9. A Visual C++ application

//Just creating the applications object runs the whole application.

```
CTheApp theApp;
```

//CMainWindow constructor

```
CMainWindow::CMainWindow()
```

```
{
```

```
LoadAccelTable("MainAccelTable")
```

```
Create(NULL,"Hello Foundation Application",
```

```
        WS_OVERLAPPEDWINDOW, rectDefault, NULL, "MainMenu");
```

```
}
```

//OnPaint

```
void CMainWindow::OnPaint()
```

```
{
```

```
CString s="Hello Windows!";
```

```
CPaintDC dc(this);
```

```
CRect rect;
```

```
GetClientRect(rect);.
```

```
dc.SetTextAlign(TA_BASELINE/TA_CENTER);
```

```
dc.SetBkMode(TRANSPARENT);
```

```
dc.TextOut((rect.right/2),(rect.bottom/2),s,s.GetLength());
```

```
}
```

Figure 6.9. Continued

```

//OnAbout

void CMainWindow::OnAbout()
{
    CModalDialog about("AboutBox, this);
    About.DoModal();
}

//Message map
BEGIN_MESSAGE_MAP(CMainWindow,CFrameWnd)
    ON_WM_PAINT()
    ON_COMMAND(IDM_ABOUT, OnAbout)
END_MESSAGE_MAP()

//CTheApp InitInstance

BOOL CTheApp::InitInstance()
{
    TRACE("HELLO WORLD\n");
    m_pMainWnd=new CMainWindow();
    m_pMainWnd->ShowWindow(m_nCmdShow);
    m_pMainWnd->UpdateWindow();
    return TRUE;
}

```

Figure 6.9. Continued

```

// hello.dlg
DLGINCLUDE RCDATA DISCARDABLE
BEGIN
    "RESOURCE.H0"
END

ABOUTBOX DIALOG 34 22,144,75
STYLE DS_MODALFRAME|WS_CAPTION|WS_SYSMENU
CAPTION "About Hello"
BEGIN
    CTEXT "Microsoft Windows",-1,0,5,144,9
    CTEXT "Microsoft Foundation Classes", -1,0,14,144,8
    CTEXT "Hello, Windows!",-1,0,23,144,8
    CTEXT "Version 1.0",0,36,144,8
    DEFPUSHBUTTON "OK",IDOK,56,53,32,14,WS_GROUP
END

//hello.rc
# include <windows.h>
# include <afxres.h>
# include "resource.h"
AFX_IDI_STD_FRAME    ICON    hello.ico

```

Figure 6.9. Continued.

```

MainMenu MENU
{
    POPUP "&Help"
    {
        MENUITEM "&About Hello....\tF1,IDM_ABOUT
    }
}

MainAccelTable ACCELERATORS
{
    VK_F1,IDM_ABOUT,VIRTKEY
}

rcinclude hello.dlg

```

Figure 6.9. Continued

Consider the above example shown which creates a single dialog box and prints “Hello Windows” onto the main screen window and includes an about dialog box. The files which assist the application here are hello.h, hello.rc, hello.cpp and hello.dlg.

Hello.rc defines the main menu and the accelerators that are used by the program’s main window. Hello.dlg defines the resource script for the Dialogbox. Hello.h has the declarations for the class interfaces used by the application and hello.cpp contains the definitions for the class functions that are available in hello.h.

The CMainWindow constructor loads the required accelerator table and creates the main window required for the application. The CTheApp::InitInstance() creates the CMainWindow and displays it. CMainWindow::OnPaint() handles the WM_PAINT messages the main window receives and when the about Dialogbox is invoked, the CMainWindow::OnAbout() is invoked to create the About Dialogbox in the application. MFC foundation classes internally traps the WM_COMMAND message with the IDM_About code and invokes the CMainWindow::OnAbout() function when required. The same applies even to the CMainWindow:: OnPaint() function. The ON_COMMAND(IDM_ABOUT ,OnAbout) maps the particular WM_COMMAND message to the OnAbout function. The whole application is started by creating an instance of CTheApp object. MFC calls the InitInstance member of the CTheApp to create and run the whole application. Figure 6.10 shows the MFC architecture.

The main things to be observed here are:

1. There is no C code involved (no event handler functions in C).
2. There is no explicit handling of messages.
3. There is no explicit message loop involved in the application.
4. The entry point to the program is hidden in the MFC class library.

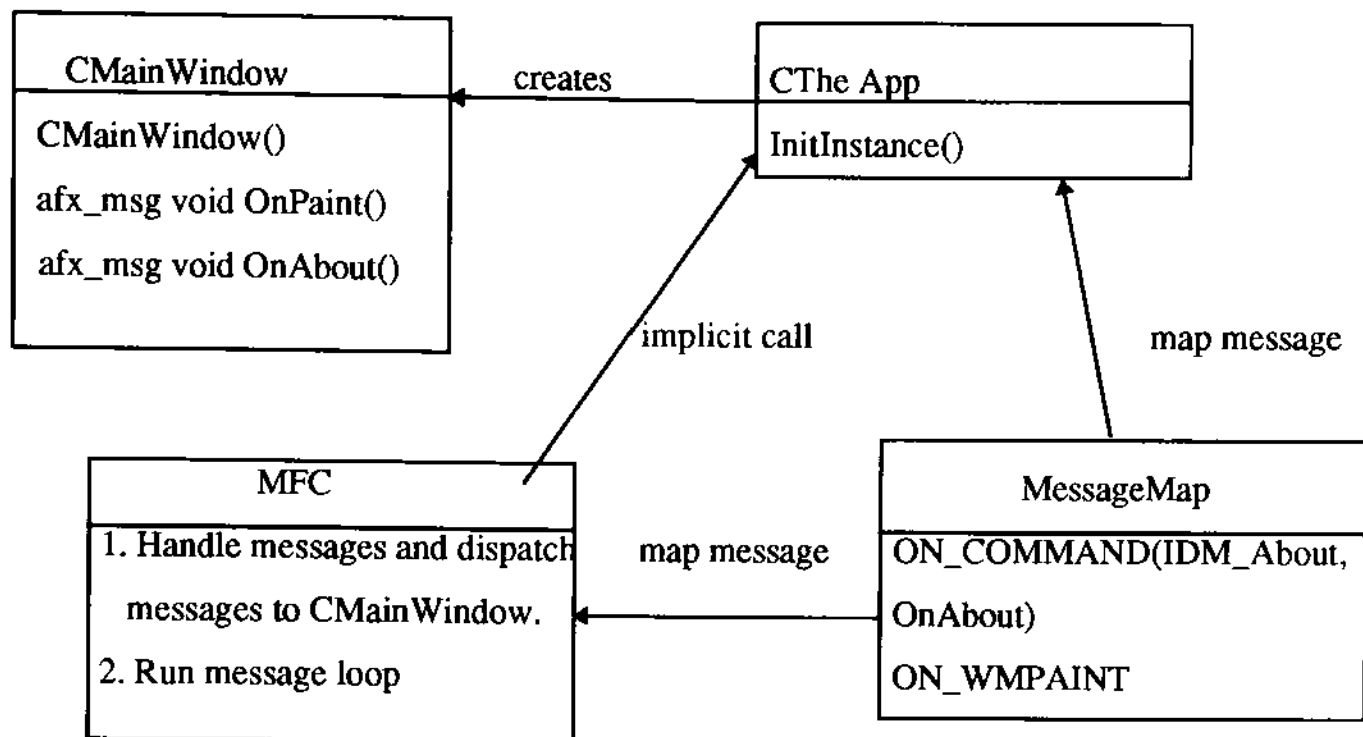


Figure 6.10. MFC architecture.

6.3.4.2 Implicit Message Handling in MFC

As mentioned before, Visual C++ follows an indirect method of handling Windows messages. All the messages are trapped and handled by MFC and the corresponding message mapped member functions are called.

For a typical Visual C++ application, the message handling MFC functions and the thread of execution are shown in Figure 6.11.

File Winmain.cpp

```
int Pascal WinMain(...)      Entry point of Windows application
{
..
if(!Afx WinInit(hInstance.....))
    goto InitFailure
    AfxGetApp()->InitInstance()    Calls the Initinstance of the
                                   Application
    AfxGetApp()->Run()    Calls the main running routine until
}}                               the application exits
```

File:appcore.cpp

```
int CWinApp::Run()
{
    ..
    for(;;)
    {
while!(:::PeekMessage(...) &&OnIdle(lIdleCount))
    {
    }
    if(!PumpMessage())
    break;
    }..}
```

Figure 6.11.MFC thread of execution

```

BOOL CWinApp::PumpMessage()
{
    .
    if(!PreTranslateMessage(...))
    {
        ::TranslateMessage(...)
        ::DispatchMessage(...)
    }
    return TRUE;
}

```

```

BOOL AFXAPI AfxWinInit(...)

```

```

{

```

```

    .

```

```

    .

```

```

if(hPrevInstance==NULL)

```

```

{ .

```

```

wndcls.lpfWndProc=AfxWndProc;   Procedure to receive messages

```

```

wndcls.hInstance=hInstance;

```

```

}}

```

Figure 6.11. continued

```
LRESULT CALLBACK Afx_EXPORT AfxWndProc(.....)
```

```
{
```

```
.
```

```
LRESULT lResult=_AfxCallWndProc()
```

```
.
```

```
}
```

```
LRESULT PASCAL _AfxCallWndProc(...)
```

```
{
```

```
.
```

```
TRY
```

```
{
```

```
lResult=pWnd->WindowProc(..)
```

```
}
```

```
CATCH_ALL(e) { }
```

```
END_CATCH_ALL
```

```
.
```

```
}
```

Figure 6.11. continued.

```
LRESULT CWnd::WindProc(....)
```

```
{  
Retrieve the message map and call the corresponding functions of the object  
class for the corresponding messages received. The window handles are  
stored in the hierarchy with specific names thus enabling easy retrieval.  
}
```

Figure 6.11. continued.

The entry point for a Windows program is `Winmain()`. Here `AfxWinInit()` is called to set the callback procedure `AfxWndProc()`. `AfxWndProc()` calls `_AfxCallWndProc()` to call `WindowProc()`. `WindowProc()` in turn retrieves the message map and handles all the messages that is received by MFC to call the corresponding message mapped functions.

6.3.4.3 Changing the Physical Interaction

The LI component in this case are the messages that have to be kept the same when porting to a different platform. The same set of messages must be operated with even on the different platform. The Physical Interaction (PI) is the toolkit component in this case which is Microsoft Windows 3.1, Windows NT, or Windows 95. If the Physical interaction has to be changed, then the entry point of the application and mapping of the virtual toolkit events have to be changed to the corresponding toolkit object events. The FC and FCA are the document classes that provide for data manipulation can remain almost the same which in this case is the

CDocument class derived from CCmdTarget. Changing the Dialog component involves changing the mode of message retrieval. In order to change the Physical Interaction, the Dialog, FC and FCA have to be modified since there are no virtual objects involved. Also, LI component has to be mapped to corresponding events on other toolkits since Visual C++ does not follow a virtual method to trap the events in case of Windows. MFC has been ported to Apple Macintosh and Motif platforms.

6.3.5 SUIT (Simple User Interface Toolkit)

A high level description of SUIT's internal architecture and basic data model is described here. A user interface toolkit requires three basic forms of support: an implementation language, operating system support and a graphics package. SUIT is implemented in ANSI-C. Pascal was not chosen since it was not powerful enough to support the external control model since it lacks the ability to store function addresses as variables. I will discuss what the external control model means later.

SUIT uses a low-level graphics package which supports graphics operations on all platforms to draw its widgets. Since very few graphics packages are implemented on all three platforms, SRGP was chosen. SRGP has been implemented on X Windows, Turbo C on DOS and even for the Macintosh. Since SRGP has several drawbacks, a graphics package called GP (Graphics Package) is provided on top of SRGP. The main objective of GP is to allow widget authors to avoid screen coordinates by providing floating point coordinate systems and perform standard window to viewport

mapping. In addition GP converts the mixed sample/event input model of SRGP into a strict event model.

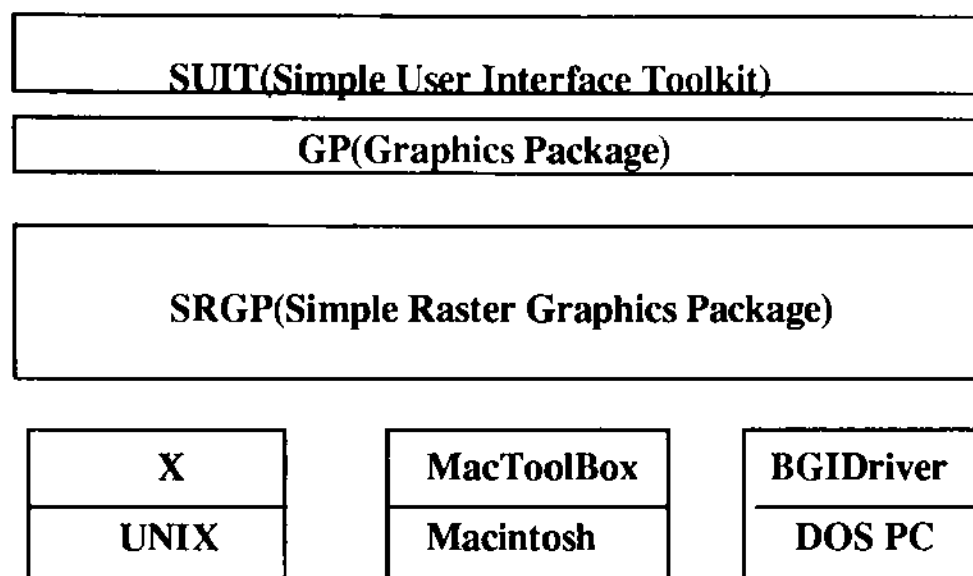


Figure 6.12. Layered software for portability (SUIT).

As explained before, SUIT follows a layered method for achieving cross platform portability.

SUIT follows a different architecture from other interface builders. The architecture of SUIT is best understood if thought of as a subroutine library that provides a front end to a database. The database maintains information about all screen objects, such as their location and color. When an application exits, SUIT writes the database to a permanent storage. The shape and position of all SUIT objects can be changed at runtime, which resides in memory. This is dumped into a file when the application exits. The next time the application is run, the database is read to get the positions and colors of all the objects on the screen.

Application programmers create objects by making calls such as
void Hi (SUIT_object obj)

```

{
printf("Hello world\n");
}

```

SUIT_createButton("hello",Hi);

This creates a button with the callback function attached to it. The callback function is invoked whenever the button is pressed. The button initially occurs at some random position. As mentioned before, it can be moved to any position required and the database is written to the disk. This is the method followed for handling callback functions to widgets.

Since SUIT uses a graphics package to draw the widgets on each platform and does not use any widgets that are supplied by an user interface environment like Windows, the widgets will look similar on all platforms.

A SUIT application is nothing but a collection of objects. This collection of objects is maintained in a table within SUIT. Each object is defined by

- Its state, contained in a property list.
- A C procedure which examines the object's state and displays it on the screen.
- A C procedure which handles user input to the object and updates the object's state.

The property list containing an object's state is a set of [name, type, value] triples as in

```

["label", "text", "pizza"]
["diameter", "floating point", 10.5].

```

It is this table that is dumped to the database file between executions of the program.

When the user input is defined by pressing both the SHIFT and CONTROL keys when the mouse button is pressed, SUII interprets the input as a command to move or resize the screen. SHIFT+CONTROL+a invokes edit and so on. This enables the programmer to change the properties of the user interface objects. The combination of SHIFT and CONTROL is practical for the only reason that this combination is rarely used by any application.

The main control loop of a SUII based application will look like [24]:
forever do

```
{  
    wait for an input event  
    if(SHIFT & CONTROL) are down  
    THEN handle as a SUII command  
    ELSE  
    {  
        figure out widget we're over and call its input handler  
    }  
    paint any objects that have changed  
}
```

The else part in the above pseudocode handles the painting and calling of the input handler after trapping an event. This mode of managing control of the widgets and event handling is called external control.

An application's main procedure looks like:

```
{  
    CreateObject(..); ...CreateObject(...)  
    letSuitRun();  
}
```

Each CreateObject() ties the callback function to the widget that is called. LetSuitRun() starts the event loop and runs till the application exits. Usually a Done button will call an exit function to terminate the program.

Support for internal control is also provided in SUIT. This can check to see if any mouse or keyboard input has occurred.

```
{    CreateObject(...);    CreateObject(...);  
    ...    while(<application's test>)  
{  
    <whatever the application wanted>  
    SUITcheckAndProcessAnyInput()  
}}
```

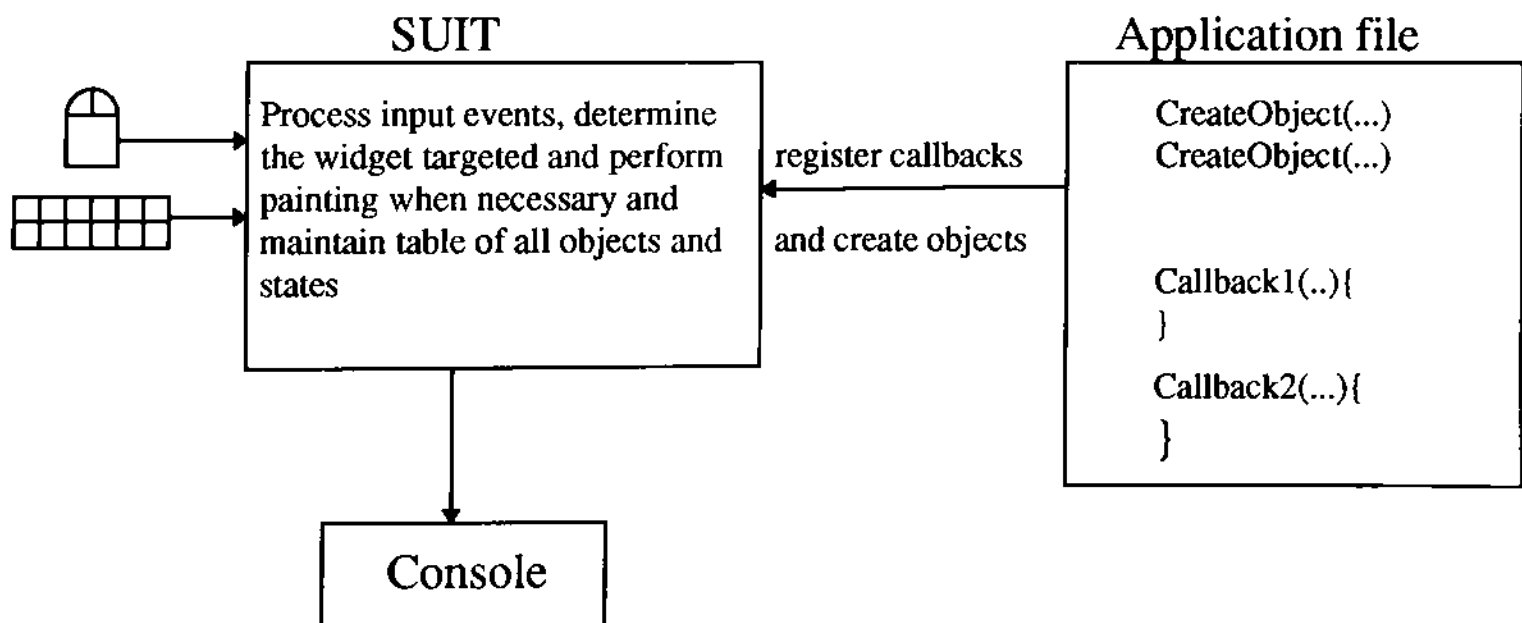


Figure 6.13. SUIT architecture.

6.3.5.1 Structure of a SUIT Application

The listing below (Figure 6.14) shows a typical SUIT application.

```
# include "suit.h"

SUIT_object num1, num2, result;

void main (int argc, char * argv[]){
    SUIT_deluxeInit(&argc,argv); init application
    SUIT_createDoneButton(NULL); create done button
    num1=-SUIT_createTypeInBox("num1",NULL); create type inbox.
    num2=SUIT_createTypeInBox("num",NULL);
    SUIT_CreateLabel("+"); create static text
    SUIT_CreateLabel("=");
    result=SUIT_CreateLabel("result"); create label
    SUIT_CreateButton("GO",PerformAddition); create button with call
                                           back
    SUIT_beginStandardApplication(NULL); begin application
}

void PerformAddition(SUIT_object object)
{
    double temp1, temp2;char buffer[100];
    temp1=atof(SUIT_getText(num1,CURRENT_VALUE)); extract
                                           number from typing box
    temp2=atof(SUIT_getText(num,2,CURRENT_VALUE);
```

Figure 6.14. A SUIT application

```
sprintf(buffer,"%lf",temp1+temp2);print result of 2 numbers  
SUIT_setText(result,LABEL,buffer); set label to result  
}
```

Figure 6.14. Continued.

The given example shows how two numbers can be obtained from the keyboard and added. Notice that there is no placement (screen coordinate) arguments anywhere. The two type in boxes and labels are created. The result is obtained by pressing the GO button. The call back PerformAddition() is tied to the GO button. When this button is pressed, SUIT automatically calls the callback procedure to obtain the result and display it on the screen.

SUIT uses the graphics package to create the canvas (main window) on which the widgets are drawn. In case of windows, one call to CreateWindow() accomplishes the job whereas for DOS, it uses the Graphics Package to create the SRGP main window.

In case of Windows, the wndclass.lpfnWndProc is set to a WndProc() function and all the SRGP events are collected and queued while in case of DOS there is a procedure to collect the mouse and keyboard events and convert it to SRGP events.

6.3.5.2 SRGP(Simple Raster Graphics Package)

A graphics application customarily divides the screen area into regions dedicated to specific purposes. When the user presses the locator button, the application must determine exactly what screen button, icon or other objects

was selected, if any, so that it can respond appropriately. This determination, called pick correlation is a fundamental part of interactive graphics.

An application program using SRGP performs pick correlation by determining in which region the cursor is located, and then an object in that region, if any, the user is selecting.

When a point is obtained from the locator (mouse), a high-level interaction-handling scheme is executed, which is essentially a dispatching procedure that uses pick correlation.

6.3.5.3 Keyboard and locator multiplexing

The scheme used for multiplexing locator and keyboard input is shown below. SRGP queues events of the devices in chronological order and SUI retrieves the first event off the queue when `SRGP_waitEvent()` is called. The procedure below allows the user to place any number of small circle markers anywhere within a rectangular drawing area. The user places a marker by pointing to the desired position and pressing button 1. The user can request that the interaction be terminated by either pressing button 3 or by typing “q” or “Q” (Figure 6.15).

```

const
PLACE =1
QUIT =3
generate initial screen layout
SRGP_setInputmode(KEYBOARD,EVENT) ;
SRGP_setKeyboardProcessingMode(RAW);
SRGP_setInputMode(LOCATOR,EVENT);
SRGP_setLocatorButtonMask([PLACE,QUIT]);{ Ignore button 2}
[Main event loop]
exit:=false
repeat
    SRGP_waitEvent(INDEFINITE,device);
    case device of
        KEYBOARD:
            begin
                SRGP_get_Keyboard(keyMeasure);
                exit:=(keyMeasure='q') or (keyMeasure ='Q')
            end;
        LOCATOR:
            begin
                SRGP_getLocator(locMeasure);
                with locMeasure do
                    case buttonofMostRecentTransition of

```

Figure 6.15. SRGP multiplexing scheme

```

PLACE:
    if(buttonChord[PLACE]=DOWN) and inDrawingArea
                                   (position) then
        SRGP_marker(position);
QUIT:
    exit:=true;
end{button case}
end{locator handling}
end{device case}
until exit

```

Figure 6.15. Continued.

The above example thus illustrates how SRGP multiplexes mouse (locator) and keyboard events within SUIT.

6.3.5.4 Changing the Physical Interaction

Figure 6.16 below shows the SUIT layered model.

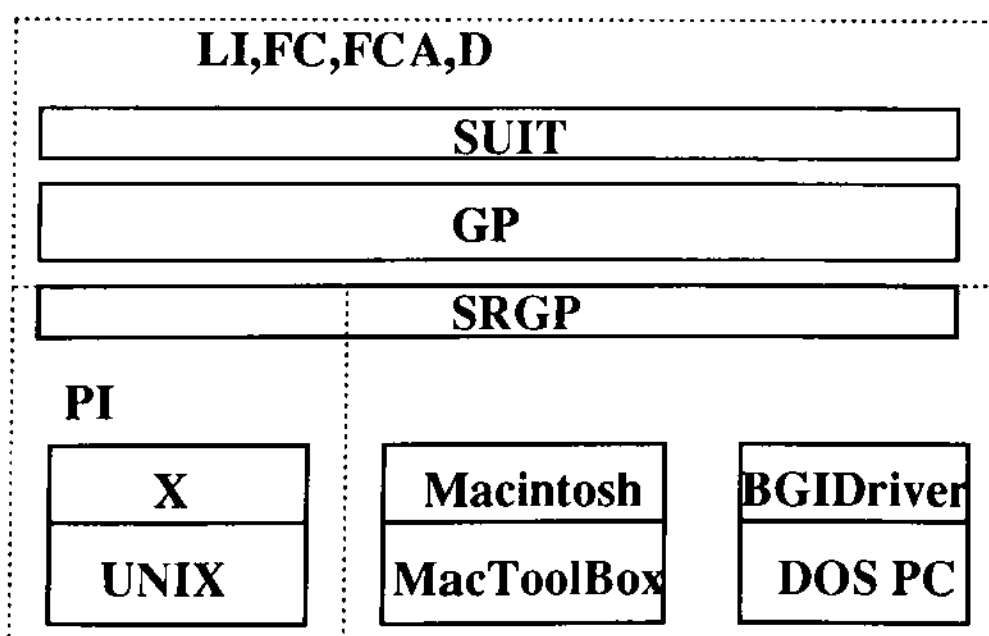


Figure 6.16. The SUIT layered model.

The SUIT and GP layer form the LI, FC, FCA and Dialog components. The SRGP component implemented for X Windows, Macintosh and DOS implement the Physical Interaction separately for each version of SRGP for that particular platform. The logical interaction comprising a set of virtual events is set separate from the physical interaction. Replacing just the SRGP library for a platform and recompiling the code on the required platform generates the application for that particular platform.

6.3.6 zApp

zApp by Inmark Inc. is a cross platform development package used to build cross platform portable user interface applications. It follows the architecture of Visual C++ but it is portable to non-Microsoft platforms like DOS Text, DOS Graphics, OS/2, X/Motif and Windows NT.

zApp follows implicit message handling like Visual C++. There is one `zApp::main()` that is called from inside the zApp library. This forms the entry point to the application. There is one set of events uniformly managed over all platforms. All the call back functions are tied to particular events that are generated as in Visual C++.

The look and feel of the application as it is ported across platforms takes the look and feel of that particular platform it is ported to. Sometimes the look is the same sometimes it may not be the case. The application code will create windows called frames and place objects called panes inside the window. Then the application will connect an event processing loop to the objects and register that loop with zApp interface. The zApp interface will

receive the native events from the environment and convert this event to zApp event format and send them to the application. The application will process the event and make requests to the environment in the form of zApp requests. The zApp interface will convert these events to the native environment event and pass that request along. By changing the zApp interface to a different platform, the application's code can remain the same. The interface extends a little further since the zApp interface will make the objects and windows have a native look within each platform. The zApp framework provides a full complement of GUI objects including

1. Windows that can be sized, moved, minimized and maximized.
2. Dialog windows containing one or more control objects.
3. List boxes, buttons, text fields and numeric controls including integers, real numbers, MDI and common dialog boxes.
4. Support for bitmaps and icons

zApp::go() starts the event loop on any platform. Depending on the event that is received by the event loop, it is converted to the native event request. The Data flow in zApp is best shown in the diagram below (Fig. 6.17).

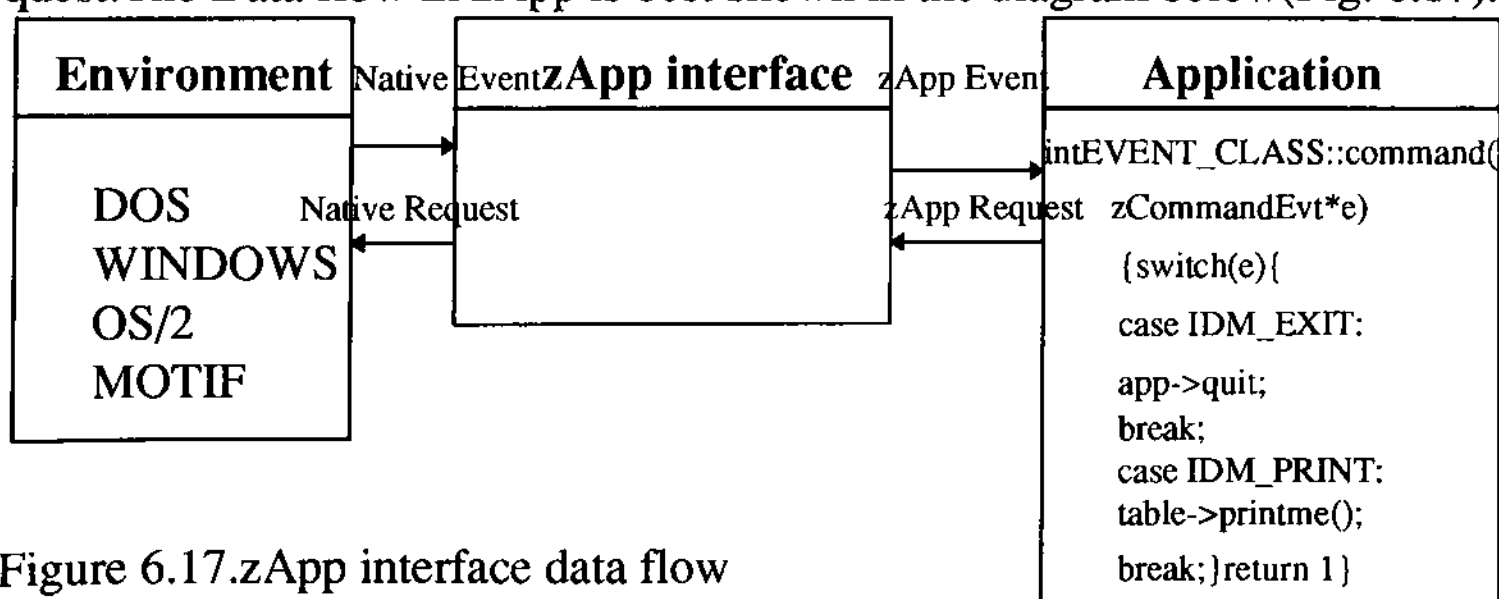


Figure 6.17.zApp interface data flow

Refer to Appendix A for the source code of the zApp application described here.

The file `tutor.cpp` contains the application's main entry point `zApp::main()`.

After the `initIntPack()` is completed, `WMain *p=ZNEW WMain` ("MDI Test App") creates the main window of the MDI Application. Here all the callback event procedures are set to the menu events. Finally all the windows are shown and updated.

In particular case where `WMain::cmdMDIWindow(zCommandEvent *e)` uses a case structure to switch between the different events and perform necessary tasks is shown in the listing in Appendix A. For DOS, zApp uses a package called MEWEL to achieve windowing. Figure 6.18 shows the zApp architecture.

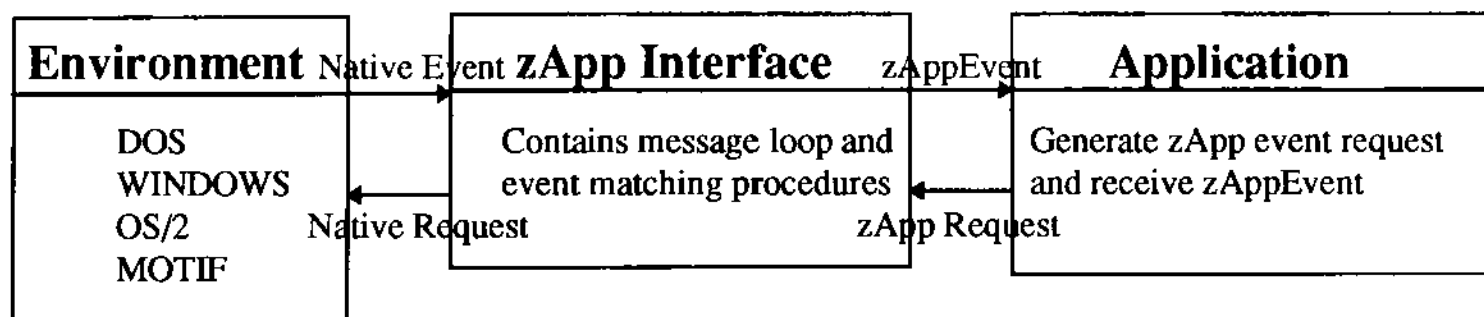


Figure 6.18. zApp architecture.

zApp has a powerful class hierarchy that supports events, Windows, DDE Events, control and so on. A portion of the hierarchy is shown (Figure 6.19). This particular application has to do with maintaining an address database of vendors and employees. The `DAddress` class (file `tutor.cpp`) derived from `zFormDialog` creates a modal dialog. The constructor is called

in `WMain::cmdDialogAddress()` to create a dialog box that accepts the addresses and personal data. This is tied to one of the menu items in the `WMain` class. The `DSearch` class (in `search.hpp`) derived from `zFormDialog` also creates a modal dialog. The constructor is called in `WMain::cmdDialogSearch()`.

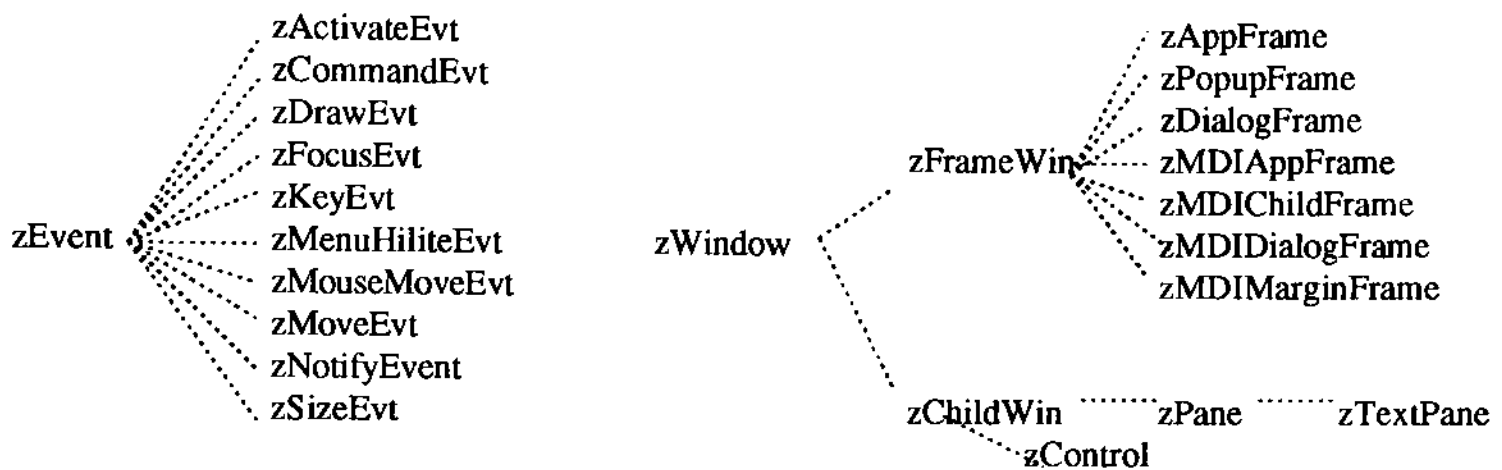


Figure 6.19. `zApp` class hierarchy

The callback procedures to be called for the menu selections are set to the member functions defined in class `WMain`. This is done using `setCommand`.

```
menu()->setCommand( this (commandProc) &WMain:: cmdFileNew,
IDMFILENEW);
```

All the resources are contained in a separate resource file of which the menu is generated using `menu(ZNEW zMenu(zResID(IDM_MainMenu1)))`;

Whenever a particular menu is pulled down or activated, the corresponding callback procedure is called and executed.

The editlines generated by calling `zEditLine(this,ID_ADDRESS, &Address,FLD_NOTREQUIRED)` and so on. This creates whatever widgets

are necessary on the Dialogbox of interest in the constructor of Dpersonal. Only the skeleton user interface code is shown here.

6.3.6.1 Changing the Physical interaction

There is a separate zApp library for each platform. Recompiling the source code with a version of the library for that particular platform generates the user interface for that particular platform.

The hierarchy contained in the zApp library is same for each platform. The entry point of the application remains the same. The message loop will be different for different platforms.

zApp follows a method of virtual events to map the generic events to the platform specific events. The events being virtual, remain the same for all platforms and forms the Logical Interaction (LI) component of the interface. As far as the Physical Interaction is concerned, changing the windowing system which forms the physical interface changes the Physical Interaction. The main thing to be remembered here is that zApp follows the method of virtual widgets. It just maps any zApp widget to a particular widget on a specific platform. The abstract interface remains the same for all the widgets and hence one set of application code. The Dialog (D) Functional Core (FC) and Functional Core Adapter (FCA) are all combined within the zApp library and will be different for different platforms.

6.4 Software architecture and steps necessary to convert the built GUI library into a cross platform library

The GUI library built for DOS and graphics comprises of a few widgets like static text, editbox, pushbutton, radiobutton, listbox, menu and popup menu. Both the libraries for DOS and Graphics are similar, with the same abstract interface. The library creates widgets by specifying the screen coordinates, widget color, background color and so on. These libraries for DOS text and DOS graphics are designed for DOS text mode and DOS Graphics mode. The interface to the library for Windows is different from that of DOS text and DOS Graphics. The library for Windows encapsulates the SDK API provided for Windows.

Our aim here is to examine the steps necessary to convert this GUI library into a cross platform library. By doing so, it must be possible to write code once on one platform and port the application to different platforms to generate the same user interface across all platforms.

The libraries are built in C++. The reason being that further extension of the library should be easy both as a GUI library and as a cross platform library. The library comprises of simple classes to generate the desired user interface across all platforms.

The header files for the classes that generate a radiobutton across all platforms are shown in Figure 6.20.

```

//include file for dos
#ifndef _RADIO_H
#define _RADIO_H
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <dos.h>
#include <stdio.h>

class radiobutton{
int x,y,bk,fr;
char *string;
public:
radiobutton(int xcoor, int ycoor, int bkclr, int frclr, char *str);
void select(void);
void deselect(void);
int get_current(void);
};

#endif

// include file for graphics
#ifndef _RADIO_H
#define _RADIO_H

```

Figure 6.20. Include files for the built library.

```

#include<stdlib.h>
#include<string.h>
#include<graphics.h>

class radiobutton{
int x,y,bk,fr,sel;
char *string;
void * buffer;
public:
radiobutton(int xcoor, int ycoor, int bkclr, int frclr, char *str);
void select(void);
void deselect(void);
int get_current(void);
};

#endif

// include file for windows
#ifndef _RADIO_H
#define _RADIO_H
#include <windows.h>
class radiobutton{
int xcoor,ycoor,id;

```

Figure 6.20.Continued.

```

HWND hwnpar;
HANDLE instan;
HWND hradbut;
char *radtext;
int cxchar,cychar,state;
public:
radiobutton(int x, int y, char *text,HWND hwnpa,HANDLE ins,int nId);
void select(void );
void deselect(void);
int get_current(void);
HWND get_handle(void);
};
#endif

```

Figure 6.20. Continued.

To create an instance of a radiobutton on DOS and Graphics, one has to just create an instance of the C++ class and use the member functions to operate on it. To use the radiobutton under windows, one has to just create the radiobutton under the WM_CREATE message and use the radiobutton's member functions whenever necessary as shown in the listing in Figure 6.21:

```

# include "app.h"
# include "text.h"
# include "button.h"
# include "checkbox.h"
# include "radio.h"
# include "editbox.h"
# include "listbox.h"
# include "menu.h"
# include "popup.h"

# include <string.h>

HANDLE instance;

int PASCAL WinMain (HANDLE hInstance,HANDLE
hPrevInstance,LPSTR lpszCmdParam,int nCmdShow)
{
instance=hInstance;
application
app("Libtest","testcap",CS_HREDRAW|CS_VREDRAW,hInstance,
hPrevInstance,nCmdShow);

}

```

Figure 6.21. Windows application

```

long FAR PASCAL _export WndProc(HWND hwnd,UINT message,UINT
wParam,LPARAM lParam)
{

static radiobutton *d;
HDC hdc;
PAINTSTRUCT ps;

switch(message)
{
case WM_CREATE:
    d= new radiobutton(20,300,"radiotext",hwnd,instance,103);
    return(0);

case WM_COMMAND:
    switch(wParam)
    {
case 103:
        if(d->get_current()==0)
            d->select();
        else
            d->deselect();
    }
}
}

```

Figure 6.21. Continued.

```

        return (0);
    }

case WM_PAINT:
    hdc=BeginPaint(hwnd,&ps);
    EndPaint(hwnd,&ps);
    return (0);

case WM_DESTROY:
    PostQuitMessage(0);
    return(0);
}

return DefWindowProc(hwnd,message,wParam,lParam);
}

```

Figure 6.21. Continued.

To convert this graphics library under all the three platforms into a cross platform library, the following steps must be taken.

1. One should have an uniform interface that is to be maintained across all the three platforms.
2. A method of implicit message handling should be maintained uniformly across all the three platforms (like in Visual C++) and a set of messages must be decided upon.
3. The entry point to the programs should be hidden within the libraries. In case of Windows, WinMain() should be hidden.

4. An interface builder must generate the necessary skeleton code necessary for the application and a visual builder must generate the resource files. The mode of generating the user interface must be decided upon beforehand. It can be the conventional visual builder where any changes to the user interface has to be changed only by the user interface builder and the application code has to be recompiled every time there is a change in the user interface. The alternate solution is to change the user interface layout on the screen at runtime. This is the method employed in SUIT. A SUIT application maintains a dynamic table internally that contains the properties of all the widgets on the screen. Whenever the screen layout is changed, the table is internally changed. A SUIT application dumps all the information contained in the table to a database file. The next time a SUIT application is run, this database is read into memory to generate the user interface at the pre-specified positions. The first time the application is run, the widgets are generated at random positions.
5. It should be possible to read the same resource files under all the platforms to generate the required widgets. To do this, the widgets have to be dynamically created after reading the resource files in case of DOS and Graphics. In case of Windows, the resources can be loaded directly from the resource file.
6. In case of DOS and Graphics where a windowing system does not exist, one must be built to keep track of all the widgets on the screen.
7. A message queuing facility has to be added under DOS and Graphics to queue all the events to the message queue.

8. The colors on different platforms can be controlled by performing platform emulation. Microsoft SDK provides functions like `GetProfileString` and `SetProfileString`, `GetProfileInt` and `SetProfileInt` to access the `WIN.INI` file settings. If these functions are rewritten for Windows and DOS to access an application initialization file to read and interpret the colors for all platforms, it is then possible to control colors across all platforms transparently.

As mentioned before, a set of events have to be decided upon and implemented on all platforms. The events and virtual widgets form the Logical Interaction (LI) part of the architecture. As far as the Physical Interaction (PI) is concerned, the widgets created in C++ using the Borland Development tools from the physical interaction for DOS text and DOS graphics. Also, the libraries developed for DOS text and DOS graphics when modified will contain the Document (D), Functional Core (FC) and Functional Core Adapter (FCA) part of the architecture.

Table 6.1 gives a comparison of all the user interface architectures examined so far and their characteristics.

Table 6.1. Comparison of architectures on different platforms.

Criterion	Serpent	Chiron	Garnet	Visual C++	Zapp	SUIT
LI and PI	not separate	separate	separate	not separate	not separate	separate
Platform	X Windows	X Windows	X Windows mac. planned	Windows 3.1 Windows NT Windows 95	UNIX, DOS Graphics, OS/2,X/Mot	DOS Text, DOS Graphics Macintosh
Runtime UI modification	no	no	no	no	no	yes
Architecture model						layered (uses package)
Method of Widgets	virtual	virtual	virtual	virtual	virtual	non-virtual

CHAPTER 7

RESULTS

One of the goals of the proposal was to develop a GUI library and examine what steps should be followed to convert this library into a cross platform library.

7.1 The developed library

7.1.1 DOS

The library which was developed in C++ was constructed with Borland C++ for DOS text screen which provided all the necessary utilities for screen manipulation. Since no graphics package was used, the library was built from a scratch. The user interface components included static text, radiobutton, checkbox, editbox, listwidget, menu, popupmenu, dialogbox etc.

The demo that is built for DOS demonstrates all the widgets in action. Figure 7.1 shows the DOS demo of the screen.

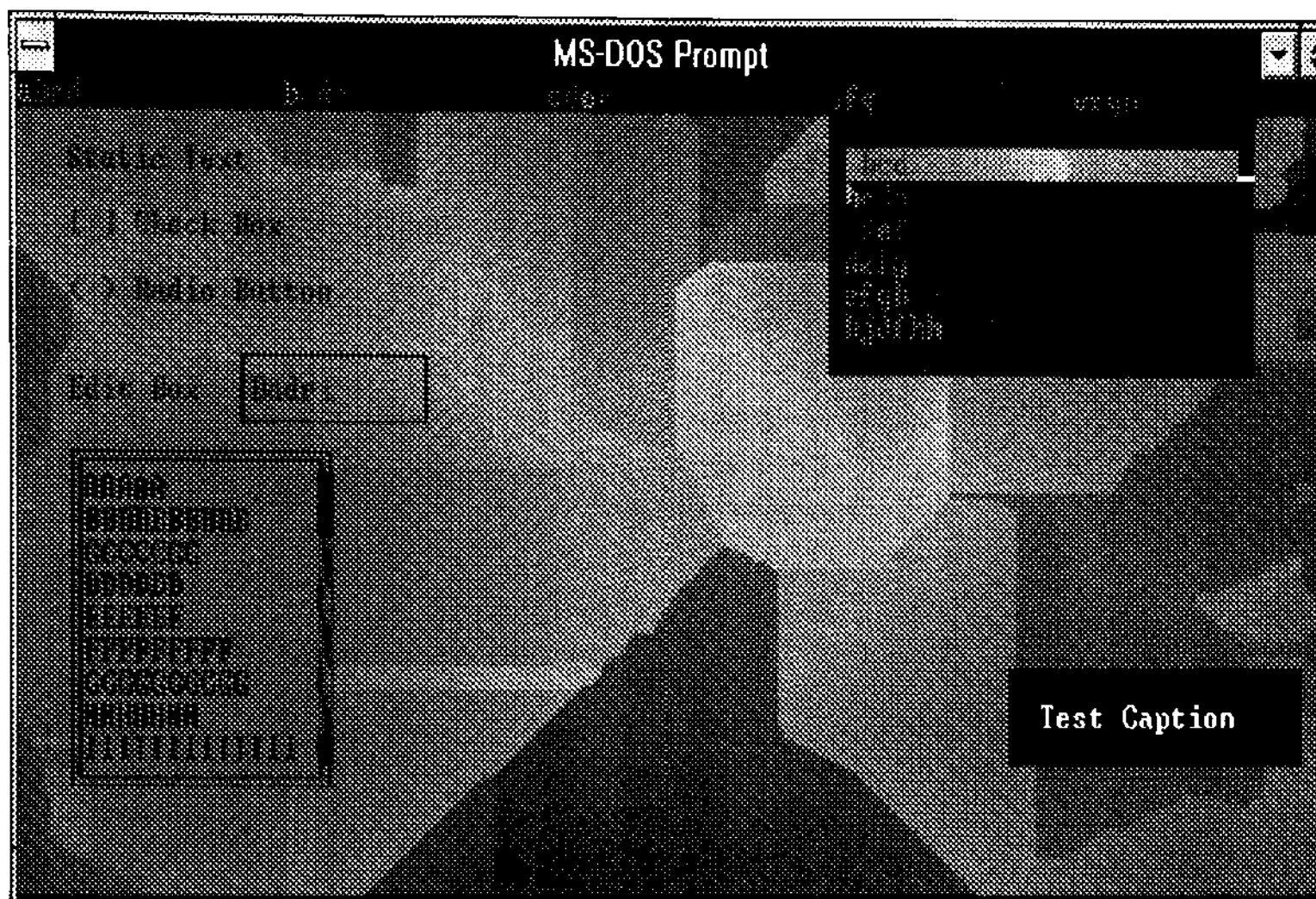


Figure 7.1. DOS demo.

7.1.2 Graphics

The same procedure was repeated for the DOS Graphics library. The Graphics mode selected was VGA-LO because of its 640x200 resolution. This makes each character on a screen a 8x8 grid and makes compatibility with the DOS text screen easy for widget placement. The library was developed with Borland C++ and contains the same list of widgets as the DOS library. Figure 7.2 shows the demo of widgets for the Graphics screen.

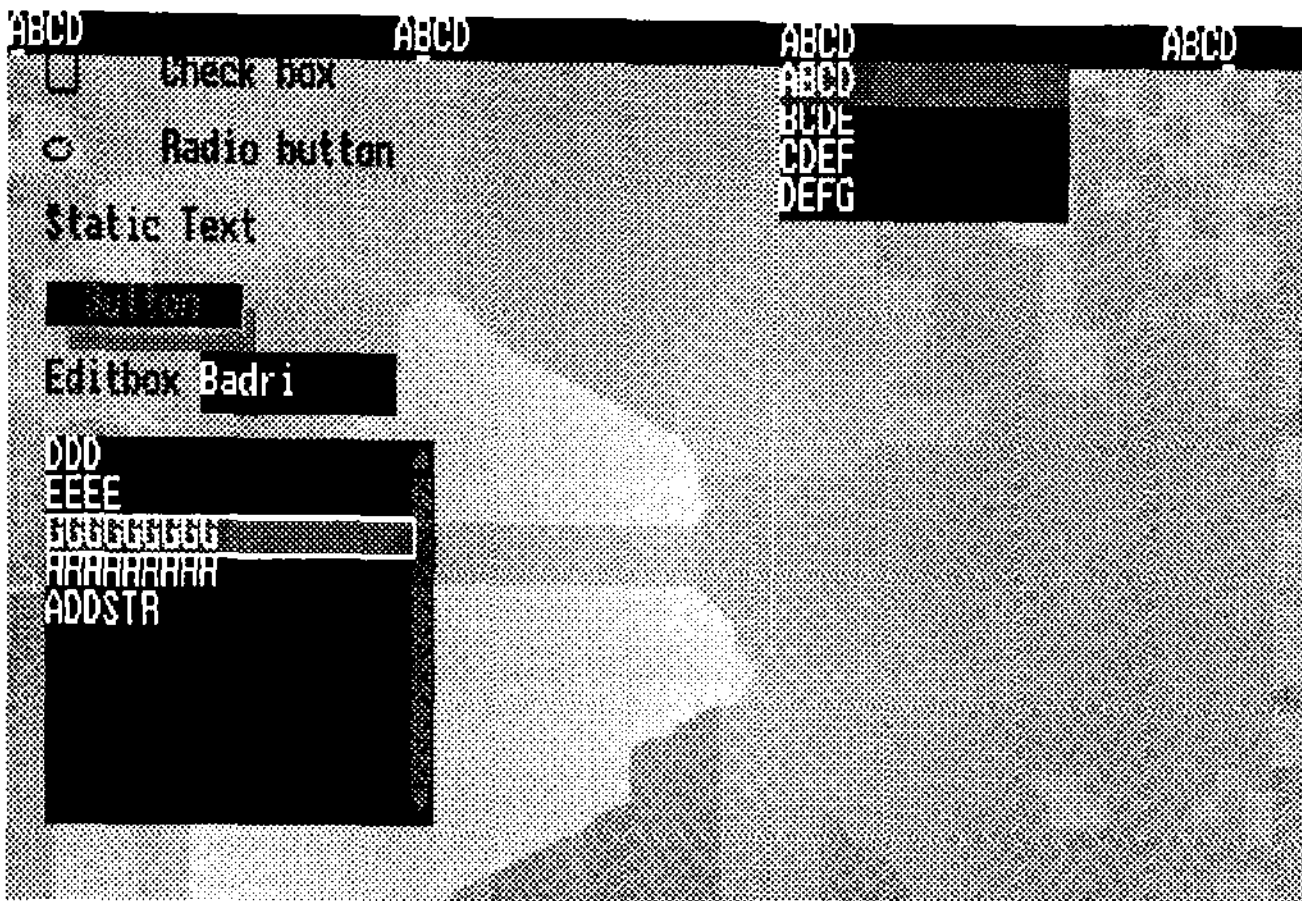


Figure 7.2. Graphics Demo

7.1.3 Windows and virtual widgets

The user interface API provided by Windows API was enclosed in C++ classes to provide a C++ interface to Windows API. The same list of widgets provided by DOS and Graphics were developed here also. Since Windows already provided the widgets in the SDK, the library only mapped the widgets in the SDK to its own. This demonstrates a case of virtual widgets.

The main disadvantages of layered software is the extent of overhead that may exist in a commercial library. In case of DOS and Graphics, the look and feel was left to the programmer. Since there is no existing

windowing system on these platforms. In case of Windows, the look and feel will conform to the look and feel of Windows platform since the library only uses the underlying API provided by Microsoft. Figure 7.3 shows the demo on windows.

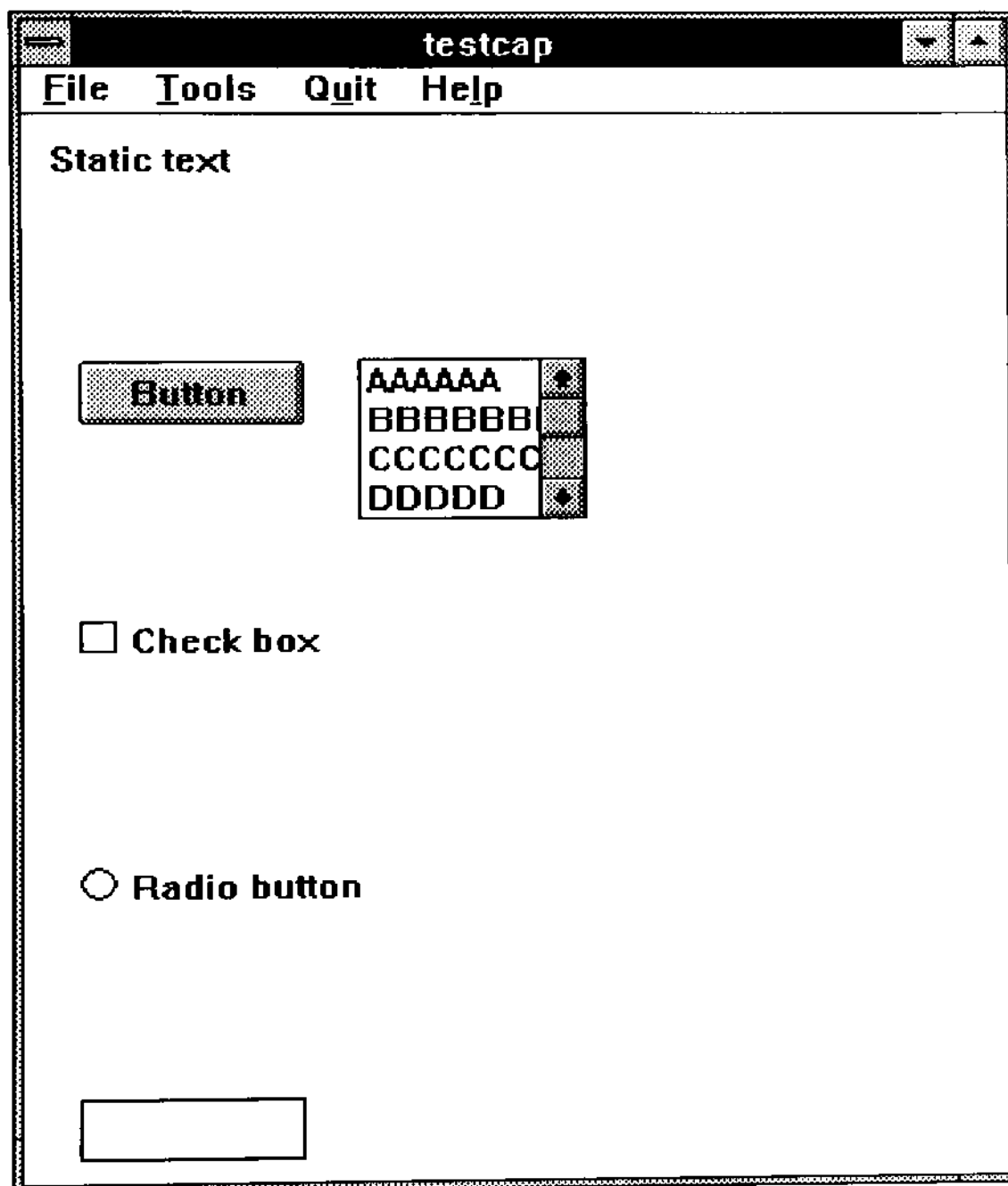


Figure 7.3. Windows demo.

7.2 SUIT and non virtual widgets

As explained before, SUIT does not use the method of virtual widgets. As explained before, SUIT uses a Graphics library on all the platforms it supports. The main observation to be made is the look and feel factors. All the SUIT applications look the same on all platforms. Hence a SUIT application will not have the same look and feel as the other applications on a platform. Since SUIT uses a layered model, the processing overhead will be more and SUIT applications always tend to be slower than other applications. The demo of SUIT demonstrates the widgets on the available platforms.

As far as converting the proposed library into a cross platform library is concerned, the cross platform library will have the following characteristics.

Table 7.1. Characteristics of the architecture for the built GUI library to be improved further

Platform	LI & PI	Cross platform resource files	Runtime UI change	UI builder	Method of widgets	Message Handling
DOS text DOS Graph. Windows	not separate	yes	no	yes	virtual	implicit

The reasons why the characteristics were chosen was explained before. To build a cross platform library, the main design considerations must be:

1. The platforms itself that the library has to support.

2. The user interface components that has to be provided on all platforms.
3. The language that the user interface has to support.
4. An uniform set of virtual messages that have to be mapped onto the actual events on all platforms.
5. Methods to handle cross platform resource files.
6. Whether the library must follow a virtual widget approach or use a graphics library to generate widgets.
7. Support for user interface builders.
8. Support for advanced features like context sensitive help.
9. Decide on any document view architecture if necessary.
10. Try to separate the Logical Interaction (LI) from the Physical interaction (PI) as far as possible.
11. Generate user interfaces on all platforms with just one set of source code.

CHAPTER 8

CONCLUSIONS AND RESEARCH DIRECTIONS

8.1 Conclusions

If a cross platform library has to be built for a particular platform, the characteristics and idiosyncrasies of the platform guide the design of the interface on the particular platform. Research should be undertaken to decide what features to support on all the platforms. Normally, all platforms have one common set of widgets.

We found that a majority of the cross platform toolkits used the method of virtual widgets for supporting user interface on all platforms.

If one wants to avoid the details of graphic primitives on all platforms, it will be better to use a graphics package for generating interfaces on all platforms. The disadvantage would be the drawbacks of the graphic package and the look and feel factors.

If C++ is used, there will be a message map procedure that associates all the events with separate callback member functions and/or a member function to handle a set of events. If C is used, there will be a callback procedures for few events and/or a single callback procedure for a set of events.

As far as resource handling is concerned, all the platforms we have encountered provide a way to handle resource files by directly loading the required resources into memory. There is always an alternate method of

creating user interface components by using the API functions the platform supports. Cross platform resource handling involves reading the resource files that follow a particular format and loading the appropriate resources into memory.

As far as API emulation on all platforms is concerned, Microsoft SDK API has a technical superiority over any other API on any other platform. This explains the reason why Motif platforms provide API emulation for Microsoft API and not vice versa. This also explains why Motif and Macintosh platforms have Visual C++ support.

The design of class hierarchy in a C++ application framework is a work of art. One thing that is always necessary and should be tried to be achieved in a cross platform library is to keep the Physical Interaction (PI) component separate from the Logical Interaction (LI) component. When this is not possible, the whole library is rewritten with an abstract interface to support cross platform code.

It is very necessary to consider and handle pick correlation in case of a cross platform library using a graphics package (like SUIT and Galaxy) to take part in event handling and screen refreshing while in the case of virtual widgets, the underlying windowing system handles pick correlation and event handling.

To summarize the research, we have started by identifying one of the major hurdles for cross platform programming which is portability. We have identified the techniques that are used in cross platform programming and the design patterns that are used if one were to program in an object oriented

language. We have analyzed the architecture of a few existing user interfaces with a software model and have examined the conditions under which it is portable to different platforms. Having studied the existing cross platform techniques for handling messages and user events and other criteria, we suggest improvements to three GUI libraries to convert it into a cross platform library. We isolated a set of design alternatives necessary to design a cross platform library.

8.2 Research Directions

A few research directions in cross platform user interfaces are listed below:

1. Cross platform resource file handling: If a resource file exists on one platform say on Microsoft Windows, it is necessary to convert the windows resource file to a resource file on any other platform say X/Motif, for the resources to be used by the application or an uniform resource file language should be developed across all platforms from which the application can read the resources.
2. Cross platform hierarchy design: Hierarchy design for a platform such as DOS, Windows and X/Motif can be very complex to provide an abstract interface for all the widgets and other operations if one wants to port cross platform code across all platforms.
3. Cross platform DLL support: DLLs (Dynamic Link Libraries) are not only specific to MS Windows platforms. Ways and means of supporting DLLs on Motif and other platforms will be an interesting topic.

4. Cross platform context sensitive help: Context sensitive help on all platforms is a very basic requirement. Techniques to support and implement cross platform context sensitive help can be further studied.
5. Cross platform DDE support: Dynamic Data Exchange (DDE) is not restricted to only Microsoft platforms. Porting such features to other platforms is a real challenge in cross platform user interfaces.
6. Voice activated user interfaces: Methods for supporting voice activated cross platform user interfaces will be a very good topic for research.
7. Malleable interfaces: This being an advanced research topic discussed in Chapter 2 under 'state of the art in user interfaces,' methods for supporting cross platform malleable interfaces can be interesting.

BIBLIOGRAPHY

- [1] Mark Andrews, 1993. *Visual C++ Object Oriented Programming*. Prentice Hall Computer Publishing, Carmel,IN.
- [2] Len Bass, Ross Faneuf and Reed Little, 1992. *A Metamodel for the Runtime Architecture of an Interactive System*, The UIMS Tool Developers Workshop, SIGCHI Bulletin, Vol. 24, no. 1, 32-37.
- [3] Bristol Technology, 1995. *Product Development Information Kit*, Ridgefield, CT.
- [4] Joëlle Coutaz, 1987. *PAC, An Object Oriented Model for Dialog Design*, Proceedings of HCI Interact, 431-436, North-Holland Publishing, Amsterdam.
- [5] Richard Chimera, 1994. *Platform Independent GUI Builders, Advance Software Engineering to Handle HCI Issues*, Human Computer Interaction Laboratory, University of Maryland, College Park, MD.
- [6] Deborah J. Mayhew, 1992. *Principles and Guidelines in Software User Interface Design*, Prentice Hall Inc., Englewood Cliffs, NJ.
- [7] *Military Standard, Defense System Software Development (DOD-STD-2167A)*, 1988. United States Department of Defense, Washington D.C.
- [8] Michael Fischer, 1991. *A Guide to Curses, A UNIX Screen Oriented Text Library*, University of Maryland, College Park, MD.

- [9] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, 1994. *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley Professional Computing Series, Addison Wesley Publishing Company, Reading, MA.
- [10] Anthony S. Glad, 1995. *Cross Platform Software Development*, Van Nostrand Reinhold, New York.
- [11] H. Rex Hartson and Deborah Boehm-Davis, 1993. *User Interface Development Processes and Methodologies*. Behaviour and Information Technology, Vol. 12, no. 2, 98-114.
- [12] John Henderson, 1988. *Software Portability*. Gower Technical Press, Gower Publications Co., Brookefield, VT.
- [13] *Inmark Demonstration Kit*, 1995. Inmark Development Coporation, Mountainview, CA.
- [14] Kaare Christian, 1992. *The Microsoft Guide to C++ programming*, Microsoft Press, Redmond, WA.
- [15] Rick Kazman, Len Bass, Gregory Abowd and Mike Webb, 1993 Oct.. *Analyzing the Properties of User Interface Software*, CMU-CS-93-201, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.
- [16] Thomas G. Lane, 1990 Nov.. *A Design Space and Design Rules for User Interface Software Architecture*, CMU/SEI-90-TR-22, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
- [17] *Product Development Information Kit*, 1995. Mainsoft Corporation, Sunnyvale, CA.

- [18] *Microsoft Foundation Class Library (MFC) Migration Kit*, 1995. Mainsoft Corporation, Bothell, WA.
- [19] Brad A. Myers and Mary Beth Roson, 1992 Feb.. *Survey on User Interface Programming*, CMU-CS-92-113, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.
- [20] Brad A Myers, 1993 July. *Why are Human User Interfaces Difficult to Design and Implement*, CMU-CS-93-183, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.
- [21] Brad A. Myers, 1994 August. *User Interface Software Tools*, CMU-CS-94-182, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.
- [22] Dan R. Olsen, James D. Foley, Scott E. Hudson, James Miller and Brad Myers, 1993. *Research directions for User Interface Software Tools*, Behaviour and Information Technology, Vol. 12, no. 2, 80-97.
- [23] Randy Pausch, Nanthaniel R. Young II and Robert Deline, 1991 Oct. *SUIT: The Pascal of User Interface Toolkits*, Computer Science Report no. TR-91-20, University of Virginia, Charlottesville, VA.
- [24] Randy Pausch, Matthew Conway and Robert Deline, 1992. *Lessons learned from SUIT, the Simple User Interface Toolkit*, University of Virginia, Charlottesville, VA.
- [25] Roger Pressman, 1992. *Software Engineering* (3rd edition), Addison-Wesley, Reading, MA.
- [26] Wade Guthrie, 1995. *An Overview of portable GUI Software*, SIGCHI Bulletin, Vol. 27, no. 1, Jan. 95.

- [27] Tomlinson, 1995. *Understanding NT, Process Portability*, Windows Developer's Journal, pp. 53-57, Oct. 95.
- [28] Peter D. Hipson, 1994. *What every Visual C++ 2 programmer should know*, SAMS publishing, Indianapolis, IN.
- [29] Brian Myers and Eric Hammer, 1994. *Windows NT programming*, Tech Publications, Singapore 0820 (Asian Edition).
- [30] *Using Visual C++ 2*, 1994. Que Corporation, Indianapolis, IN.
- [31] Garlan, D. and Shaw, M. 1993. *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing, Singapore.
- [32] Sommerville, I., 1992. *Software Engineering*, 4th edition, Addison-Wesley Publishing, Reading, MA.
- [33] Bass, L., Clapper, B., Hardy E., Kazman, R. and Seacord, R., 1990. *Serpent: A User Interface Management System*. Proceedings of the Winter 1990 USENIX Conference, Berkeley, CA. 245-258.
- [34] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. 1996. *Computer Graphics: Principles and Practice* (second edition), Addison Wesley Publishing Company, Reading, MA.

APPENDIX A

STRUCTURE OF A zApp APPLICATION

```
//tutor.h

#ifdef IDHELP
#undef IDHELP
#endif

#define IDHELP          998

#define IDM_MDIWindowMenu  500

#define IDD_Address       501
#define IDD_Search       502
#define IDD_About        503
#define IDD_Personal     504
#define IDI_Main         505
#define IDI_Personal     506
#define IDB_BmpViewBitmap1 507
#define IDB_MainTools    508
#define IDM_CASCADE      751
#define IDM_TILE         752
#define IDM_ARRANGEICONS 753
#define IDM_MainMenu1    800
#define IDM_FILE         801
#define IDM_FILENEW      802
#define IDM_FILEOPEN     803
```

#define IDM_FILESAVE	804
#define IDM_FILESAVEAS	805
#define IDM_FILEPRINT	806
#define IDM_FILEEXIT	807
#define IDM_EDIT	808
#define IDM_EDITCUT	809
#define IDM_EDITCOPY	810
#define IDM_EDITPASTE	811
#define IDM_EDITLEFT	812
#define IDM_EDITCENTER	813
#define IDM_EDITRIGHT	814
#define IDM_DIALOGS	815
#define IDM_DIALOGSADDRESS	816
#define IDM_DIALOGSSEARCH	817
#define IDM_DIALOGSMDIDIALOG	818
#define IDM_HELP	819
#define IDM_HELPCONTENTS	820
#define IDM_HELPSEARCH	821
#define IDM_HELPABOUT	822
#define ID_COMBOBOX1	100
#define ID_MALE	101
#define ID_FEMALE	102
#define ID_EDIT1	103
#define ID_EDIT2	104

#define ID_EDIT3	105
#define ID_EDIT4	106
#define ID_EDIT5	107
#define ID_EDIT6	108
#define ID_HOMENUM	109
#define ID_FAXNUM	110
#define ID_SSN	111
#define ID_EDIT7	112
#define ID_SEARCH	113
#define ID_HELP	114
#define ID_LISTBOX1	115
#define ID_FIRSTNAME	116
#define ID_MI	117
#define ID_LASTNAME	118
#define ID_SOCSEC	119
#define ID_ADDRESS	120
#define ID_CITY	121
#define ID_STATE	122
#define ID_ZIP	123
#define ID_TELEPHONE	124
// zpb_begin AdditionalIDs	
// zpb_end	

```

//
// tutor.hpp - MDI Sample Application
// Version - 1.0
// 5/13/94 - 4:31:30pm
//
//
// zpb_begin Revisions
// zpb_end

#include <zapph/mdi.hpp>
#include <intpack/zip.hpp> // Interface Pack
#include "tutor.h" // Resource Constants

// zpb_begin AppIncludes
// zpb_end
//tutor.hpp
#include "address.hpp" // Module Class Definitions
#include "search.hpp" // Module Class Definitions

typedef int (zEvH::*NotifyEvtProc)(zNotifyEvt *);
typedef int (zEvH::*NotifyFocusProc)(zFocusEvt *);
void centerWindow(zWindow *w, BOOL fOnParent = TRUE);

class zfBitmapPane : public zPane {

```

```

        zBitmap theBitmap;
public:
        zfBitmapPane(zWindow* w, const zResId& id, zSizer* sz);
        virtual int draw(zDrawEvt *e);
};

class tbMainMainTools : public zToolbar {
        zBitmap *theBmp;
        zButtonGroup *grpEditLeft;
public:
        zStaticText *pStatic1;
        zComboBoxBase *pComboBox1;
        tbMainMainTools(zMDIMarginFrame *w, zSizer *sz, zBitmap
        *bmp=0);
        ~tbMainMainTools();

        // zpb_begin ptbMainMainToolsClass
        // zpb_end
};

class WMain : public zMDIAppFrame {
public:
        zToolbar *pTB;

```

```
WMain(const char* title);
~WMain();

int cmdFileNew(zCommandEvt*);
int cmdFileOpen(zCommandEvt*);
int cmdFileSave(zCommandEvt*);
int cmdFileSaveas(zCommandEvt*);
int cmdFilePrint(zCommandEvt*);
int cmdFileExit(zCommandEvt*);
int cmdEditCut(zCommandEvt*);
int cmdEditCopy(zCommandEvt*);
int cmdEditPaste(zCommandEvt*);
int cmdEditLeft(zCommandEvt*);
int cmdEditCenter(zCommandEvt*);
int cmdEditRight(zCommandEvt*);
int cmdDialogsAddress(zCommandEvt*);
int cmdDialogsSearch(zCommandEvt*);
int cmdDialogsMDIIdialog(zCommandEvt*);
int cmdHelpContents(zCommandEvt*);
int cmdHelpSearch(zCommandEvt*);
int cmdHelpAbout(zCommandEvt*);
int cmdMDIWindow(zCommandEvt*);

// zpb_begin WMainClass
// zpb_end
```

```
};
```

```
class WBmpView : public zMDIChildFrame {
```

```
public:
```

```
    WBmpView(zMDIAppFrame *w, const char *title);
```

```
    ~WBmpView();
```

```
    // zpb_begin WBmpViewClass
```

```
    // zpb_end
```

```
};
```

```
class DAbout : public zFormDialog {
```

```
    zButton *pOK;
```

```
public:
```

```
    DAbout(zWindow *, const zResId&);
```

```
    // zpb_begin DAboutClass
```

```
    // zpb_end
```

```
};
```

```
class DPersonal : public zMDIFormDialog {
```

```
    zEditLine *pFirstname, *pMi, *pLastname, *pAddress, *pCity,  
    *pState;
```

```

        zStringEdit *pSocsec, *pZip, *pTelephone;
        zButton *pOk, *pCancel;
public:
        zString _Firstname, _Mi, _Lastname, _Address, _City, _State;
        zString _Socsec, _Zip, _Telephone;
        DPersonal(zMDIAppFrame *, const zResId&);

        // zpb_begin DPersonalClass
        // zpb_end
};

//
// tutor.cpp - MDI Sample Application
// Version - 1.0
// 5/13/94 - 4:31:38pm
//
//
// zpb_begin AppRevisions
// zpb_end

#include "zapp.hpp"
#include "tutor.hpp"      // Main App Class Definitions & Includes
ZAPP_IMPL_ASSERTS;

```

```

// zpb_begin GlobalVars
// zpb_end

//
//      Initial data for list & combo boxes for Main
//
char *MainToolsComboBox1Data[] = {
    "Arial",
    "Helv",
    0
};
// ToolBar Member Functions - MainTools
tbMainMainTools::tbMainMainTools(zMDIMarginFrame *w, zSizer *sz,
zBitmap *bmp): zToolbar(w, sz)
{theBmp = bmp;
pStatic1 = ZNEW zStaticText(this, ZNEW zSizer(zDialogUnit(152,2),
DialogUnit(20,8)), SS_LEFT, "Font:", -1);
pStatic1->backgroundColor(zColor(LTGRAY));
pStatic1->show();

    pComboBox1 = ZNEW zComboBox(this, ZNEW
zSizer(zDialogUnit(174,0),zDialogUnit(48,35)),
WS_TABSTOP|CBS_DROPDOWNLIST, "", ID_COMBOBOX1);
pComboBox1->addCharStrings(MainToolsComboBox1Data);

```

```

pComboBox1->selection(0);
pComboBox1->show();
(ZNEW zToolButton(this, ZNEW zSizer(zPoint(4,1), zDimension(24,22)),
0, IDM_FILENEW, theBmp, zRect(0, 0, 16, 15)))>show();
(ZNEW zToolButton(this, ZNEW zSizer(zPoint(27,1),
zDimension(24,22)),0, IDM_FILEOPEN, theBmp, zRect(16, 0, 32, 15)))
->show();
(ZNEW zToolButton(this, ZNEW zSizer(zPoint(50,1),
zDimension(24,22)),0, IDM_FILESAVE, theBmp, zRect(32, 0, 48, 15)))
->show();
(ZNEW zToolButton(this, ZNEW zSizer(zPoint(73,1),
zDimension(24,22)),0, IDM_FILEPRINT, theBmp, zRect(48, 0, 64, 15)))
->show();
(ZNEW zToolButton(this, ZNEW zSizer(zPoint(105,1),
zDimension(24,22)),0, IDM_EDITCUT, theBmp, zRect(64, 0, 80, 15)))
->show();
(ZNEW zToolButton(this, ZNEW zSizer(zPoint(128,1), zDimension(24,22)),
0, IDM_EDITCOPY, theBmp, zRect(80, 0, 96, 15)))
->show();
(ZNEW zToolButton(this, ZNEW zSizer(zPoint(151,1), zDimension(24,22)),
0, IDM_EDITPASTE, theBmp, zRect(96, 0, 112, 15)))
->show();
(ZNEW zToolButton(this, ZNEW zSizer(zPoint(183,1), zDimension(24,22)),
0|ZBB_TWOSTATE, IDM_EDITLEFT, theBmp, zRect(112, 0, 128, 15)))

```

```

        ->show();

(ZNEW zToolButton(this, ZNEW zSizer(zPoint(206,1), zDimension(24,22)),
0|ZBB_TWOSTATE, IDM_EDITCENTER, theBmp, zRect(128, 0, 144,
15)))

        ->show();

(ZNEW zToolButton(this, ZNEW zSizer(zPoint(229,1), zDimension(24,22)),
0|ZBB_TWOSTATE, IDM_EDITRIGHT, theBmp, zRect(144, 0, 160, 15)))

        ->show();

(ZNEW zToolButton(this, ZNEW zSizer(zPoint(260,1), zDimension(24,22)),
0, IDM_HELPCONTENTS, theBmp, zRect(160, 0, 176, 15)))

        ->show();

grpEditLeft = ZNEW zButtonGroup(this, IDM_EDITLEFT,
IDM_EDITRIGHT);

        // zpb_begin tbMainMainToolsConstructor
        // zpb_end
}

tbMainMainTools::~~tbMainMainTools() {
        delete grpEditLeft;
        if (theBmp)
                delete theBmp;
}

//

```

```

// Frame Member Functions - Main
//
// Window Constructor
WMain::WMain(const char *title)
: zMDIAppFrame(0,ZNEW zSizer(),zSTDFRAME,
    title, ZNEW zMenu(zResId(IDM_MDIWindowMenu))) {
    menu(ZNEW zMenu(this, zResId(IDM_MainMenu1)));
    menu()->insertDropDown(MDIImenu(),"&Window", 3);
    menu()->setCommand(this,
(CommandProc)&WMain::cmdMDIWindow, IDM_CASCADE,
IDM_ARRANGEICONS);
    menu()->setCommand(this, (CommandProc)&WMain::cmdFileNew,
IDM_FILENEW);
    menu()->setCommand(this, (CommandProc)&WMain::cmdFileOpen,
IDM_FILEOPEN);
    menu()->setCommand(this, (CommandProc)&WMain::cmdFileSave,
IDM_FILESAVE);
    menu()->setCommand(this, (CommandProc)&WMain::cmdFileSaveas,
IDM_FILESAVEAS);
    menu()->setCommand(this, (CommandProc)&WMain::cmdFilePrint,
IDM_FILEPRINT);
    menu()->setCommand(this, (CommandProc)&WMain::cmdFileExit,
IDM_FILEEXIT);

```

```

        menu()->setCommand(this, (CommandProc)&WMain::cmdEditCut,
IDM_EDITCUT);

        menu()->setCommand(this, (CommandProc)&WMain::cmdEditCopy,
IDM_EDITCOPY);

        menu()->setCommand(this, (CommandProc)&WMain::cmdEditPaste,
IDM_EDITPASTE);

        menu()->setCommand(this, (CommandProc)&WMain::cmdEditLeft,
IDM_EDITLEFT);

        menu()->setCommand(this, (CommandProc)&WMain::cmdEditCenter,
IDM_EDITCENTER);

        menu()->setCommand(this, (CommandProc)&WMain::cmdEditRight,
IDM_EDITRIGHT);

        menu()->setCommand(this,
(CommandProc)&WMain::cmdDialogsAddress,
IDM_DIALOGSADDRESS);

        menu()->setCommand(this,
(CommandProc)&WMain::cmdDialogsSearch, IDM_DIALOGSSEARCH);

        menu()->setCommand(this,
(CommandProc)&WMain::cmdDialogsMDIdialog,
IDM_DIALOGSMDIDIALOG);

        menu()->setCommand(this,
(CommandProc)&WMain::cmdHelpContents, IDM_HELPCONTENTS);

        menu()->setCommand(this,
(CommandProc)&WMain::cmdHelpSearch, IDM_HELPSEARCH);

```

```

        menu()->setCommand(this, (CommandProc)&WMain::cmdHelpAbout,
IDM_HELPABOUT);

        setIcon(ZNEW zIcon(zResId(IDI_Main)));

        // Create Tool Bar Margin Frame

        zMDIMarginFrame *ToolFrame = ZNEW zMDIMarginFrame(this,
                ZNEW zGrowToFitSizer(ZGRAV_TOP, sizer()));

        ToolFrame->show();

        pTB = ZNEW tbMainMainTools(ToolFrame, ZNEW
zGravSizer(ZGRAV_TOP, 26, ToolFrame->sizer()), ZNEW
zBitmap(zResId(IDB_MainTools)));

        pTB->show();

        zMDIMarginFrame *sf=ZNEW zMDIMarginFrame(this,ZNEW
zGrowToFitSizer(ZGRAV_BOTTOM,sizer()));

        zStatusLineEZ* pSB = ZNEW zStatusLineEZ(sf, ZNEW
zGravSizer(ZGRAV_BOTTOM,0, sf->sizer()), ZSL_STANDARDITEM);

        sf->show();

        pSB->show();

        sizer()->update();

        // zpb_begin WMainConstructor

        // zpb_end

        show();

}

```

```

WMain::~~WMain() {

```

```

        // zpb_begin WMainDestructor1
        // zpb_end
    }

    //
    // Menu Item Selection Handlers
    //
    int WMain::cmdMDIWindow(zCommandEvt* ev) {
        // zpb_begin WMainMDICmds
        switch (ev->cmd()) {
            case IDM_CASCADE:
                cascade();
                break;
            case IDM_TILE:
                tile();
                break;
            case IDM_ARRANGEICONS:
                arrangeIcons();
                break;
        }
        // zpb_end
        return 1;
    }

```

```

int WMain::cmdFileNew(zCommandEvent* ev) {
    ZNEW WBmpView(this, "MDI Child Win");
    // zpb_begin WMainFileNew
    // zpb_end
    return 1;
}

int WMain::cmdFileOpen(zCommandEvent* ev) {
    // zpb_begin WMainFileOpenOpenDlg
    char *types[] = {
        "All Files (*.*)", "*.*",
        "Text Files (*.txt)", "*.txt",
        0,0
    };
    zFileOpenForm *p = ZNEW zFileOpenForm(this, "Open File", 0,
types);
    if (p->completed()) {
        // Use p->name() to retrieve filename
    }
    delete p;
    //zpb_end

    // zpb_begin WMainFileOpen
    // zpb_end

```

```

        return 1;
    }

int WMain::cmdFileSave(zCommandEvent* ev) {
    // zpb_begin WMaincmdFileSave
    // zpb_end
    return 1;
}

int WMain::cmdFileSaveas(zCommandEvent* ev) {
    // zpb_begin WMainFileSaveasSaveDlg
    char *types[] = {
        "All Files (*.*)", "*.*",
        "Text Files (*.txt)", "*.txt",
        0, 0
    };
    zFileSaveAsForm *p = ZNEW zFileSaveAsForm(this, "Save File As",
0, types);
    if (p->completed()) {
        // Use p->name() to retrieve filename
    }
    delete p;
    //zpb_end
}

```

```

        // zpb_begin WMainFileSaveas
        // zpb_end
        return 1;
    }

int WMain::cmdFilePrint(zCommandEvent* ev) {
    // zpb_begin WMainFilePrintPrintDlg
    zPrinterDisplay *pr=ZNEW zPrinterDisplay;
    if (!pr->isValid()) {
        zMessage msg(this,"No Printer drivers installed","Unable to print");
        return 1;
    }
    if (pr->printerSetup()) {
    }
    delete pr;
    //zpb_end

    // zpb_begin WMainFilePrint
    // zpb_end
    return 1;
}

int WMain::cmdFileExit(zCommandEvent* ev) {
    // zpb_begin WMainFileExit

```

```

        // zpb_end
        app->quit();
        return 1;
    }

int WMain::cmdEditCut(zCommandEvent* ev) {
    // zpb_begin WMaincmdEditCut
    // zpb_end
    return 1;
}

int WMain::cmdEditCopy(zCommandEvent* ev) {
    // zpb_begin WMaincmdEditCopy
    // zpb_end
    return 1;
}

int WMain::cmdEditPaste(zCommandEvent* ev) {
    // zpb_begin WMaincmdEditPaste
    // zpb_end
    return 1;
}

int WMain::cmdEditLeft(zCommandEvent* ev) {

```

```
    // zpb_begin WMaincmdEditLeft
    // zpb_end
    return 1;
}
```

```
int WMain::cmdEditCenter(zCommandEvt* ev) {
    // zpb_begin WMaincmdEditCenter
    // zpb_end
    return 1;
}
```

```
int WMain::cmdEditRight(zCommandEvt* ev) {
    // zpb_begin WMaincmdEditRight
    // zpb_end
    return 1;
}
```

```
int WMain::cmdDialogsAddress(zCommandEvt* ev) {
    DAddress* p=ZNEW DAddress(this, zResId(IDD_Address));
    p->modal();
    if (p->completed()) {
        // zpb_begin WMainDialogsAddress
        // zpb_end
    }
}
```

```

        delete p;
        return 1;
    }

int WMain::cmdDialogsSearch(zCommandEvt* ev) {
    DSearch* p=ZNEW DSearch(this, zResId(IDD_Search));
    p->modal();
    if (p->completed()) {
        // zpb_begin WMainDialogsSearch
        // zpb_end
    }
    delete p;
    return 1;
}

```

```

int WMain::cmdDialogsMDIdialog(zCommandEvt* ev) {
    DPersonal* p=ZNEW DPersonal(this, zResId(IDD_Personal));
    p->deleteOnClose(TRUE);
    // zpb_begin WMainDialogsMDIdialog
    // zpb_end
    return 1;
}

```

```

int WMain::cmdHelpContents(zCommandEvt* ev) {

```

```

        // zpb_begin WMainHelpContents
        // zpb_end
        return 1;
    }

int WMain::cmdHelpSearch(zCommandEvent* ev) {
    // zpb_begin WMainHelpSearch
    // zpb_end
    return 1;
}

int WMain::cmdHelpAbout(zCommandEvent* ev) {
    DAbout* p=ZNEW DAbout(this, zResId(IDD_About));
    p->modal();
    if (p->completed()) {
        // zpb_begin WMainHelpAbout
        // zpb_end
    }
    delete p;
    return 1;
}

// zpb_begin WMainMemberFunctions
// zpb_end

```

```

//
// Frame Member Functions - BmpView
//
// Window Constructor
WBmpView::WBmpView(zMDIAppFrame *w, const char *title)
: zMDIChildFrame(w,ZNEW zSizer(0,30,376,205),zSTDFRAME, title) {
    deleteOnClose(TRUE);
    backgroundColor(zColor(64,128,128));
    ZNEW zfBitmapPane(this, zResId(IDB_BmpViewBitmap1), ZNEW
zSizer(73,41,225,125));
    sizer()->update();
    // zpb_begin WBmpViewConstructor
    // zpb_end
    show();
}

WBmpView::~WBmpView() {
    // zpb_begin WBmpViewDestructor1
    // zpb_end
}

// zpb_begin WBmpViewMemberFunctions

```

```

// zpb_end

//
// Dialog Member Functions - About
//
DAbout::DAbout(zWindow *w,const zResId& rid) : zFormDialog(w,rid) {
    sizer()->update();
    // zpb_begin DAboutConstructor
    // zpb_end
    centerWindow(this);    // Center window on parent
    show();
}

// zpb_begin DAboutMemberFunctions
// zpb_end


//
// Dialog Member Functions - Personal
//
DPersonal::DPersonal(zMDIAppFrame *w,const zResId& rid) :
zMDIFormDialog(w,rid) {
    deleteOnClose(TRUE);

```

```

        setIcon(ZNEW zIcon(zResId(IDI_Personal)));

        pFirstname = ZNEW zEditLine(this, ID_FIRSTNAME, &_Firstname,
FLD_NOTREQUIRED);

        pMi = ZNEW zEditLine(this, ID_MI, &_Mi, FLD_NOTREQUIRED);

        pLastname = ZNEW zEditLine(this, ID_LASTNAME, &_Lastname,
FLD_NOTREQUIRED);

        pSocsec = ZNEW zStringEdit(this, ID_SOCSEC, &_Socsec, "999'-
'99'-'9999", FLD_NOTREQUIRED);

        pAddress = ZNEW zEditLine(this, ID_ADDRESS, &_Address,
FLD_NOTREQUIRED);

        pCity = ZNEW zEditLine(this, ID_CITY, &_City,
FLD_NOTREQUIRED);

        pState = ZNEW zEditLine(this, ID_STATE, &_State,
FLD_NOTREQUIRED);

        pZip = ZNEW zStringEdit(this, ID_ZIP, &_Zip, "99999",
FLD_NOTREQUIRED);

        pTelephone = ZNEW zStringEdit(this, ID_TELEPHONE,
&_Telephone, "'(999)' '999'-'9999", FLD_NOTREQUIRED);

        sizer()->update();

        // zpb_begin DPersonalConstructor

        // zpb_end

        show();

}

```

```

// zpb_begin DPersonalMemberFunctions
// zpb_end


//
// Simple function to center window within parent or screen
//
void centerWindow(zWindow *w, BOOL fOnParent) {
    zRect winRect, parentRect;
    int xWin, yWin;

    w->getExterior(winRect);
    zSystemInfo screen;

    if (fOnParent && w->parent()) {
        // retrieve parent rectangle
        w->parent()->getExterior(parentRect);

        // center within parent window
        xWin = parentRect.left() + ((parentRect.width() -
winRect.width())/2);
        yWin = parentRect.top() + ((parentRect.height() -
winRect.height())/2);
    }
}

```

```

        // adjust win x-location for screen size
        if ( xWin+winRect.width() > screen.pixWidth() )
            xWin = screen.pixWidth() - winRect.width();

        // adjust win y-location for screen size
        if ( yWin+winRect.height() > screen.pixHeight() )
            yWin = screen.pixHeight() - winRect.height();
    }
    else {
        // center within entire screen
        xWin = (screen.pixWidth() - winRect.width()) / 2;
        yWin = (screen.pixHeight() - winRect.height()) / 2;
    }

    // move window to new location
    w->move( (xWin>0) ? xWin : 0,
            (yWin>0) ? yWin : 0,
            winRect.width(),
            winRect.height());
}

//
// Bitmap Pane Member Functions
//

```

```
zfBitmapPane::zfBitmapPane(zWindow* w, const zResId &id, zSizer* sz)
    :    zPane(w, sz), theBitmap(id) {
    show();
}
```

```
int zfBitmapPane::draw(zDrawEvt *e) {
    zRect r;
    getInterior(r);           // Draw entire bitmap
    canvas()->lock();

    zBitmapDisplay *bd;
    bd = ZNEW zBitmapDisplay(&theBitmap);
    bd->lock();
    bd->copyTo(canvas(), r.left(), r.top(),
               theBitmap.size().width(), theBitmap.size().height(), 0, 0);
    bd->setBitmap(0);
    bd->unlock();
    delete bd;
    canvas()->unlock();
    return TRUE;
}
```

```
// zpb_begin AppUserCode
// zpb_end
```

```

//
// Application Entry Point
//
void zApp::main() {
    initIntPack();

    // zpb_begin AppMain
    // zpb_end

    WMain* p=ZNEW WMain("MDI Test App");
    // zpb_begin AppMain2
    // zpb_end

    go();

    delete p;

    // zpb_begin AppMain3
    // zpb_end
}

//address.hpp
class DAddress : public zFormDialog {
    zEditLine *pEdit1, *pEdit2, *pEdit3, *pEdit4, *pEdit5;
    zEditBox *pEdit7;
    zStringEdit *pEdit6, *pHomeNum, *pFaxNum, *pSSN;
    zComboBox *pComboBox1;

```

```

        zRadioButton *pMale, *pFemale;
        zButton *pOk, *pCancel, *pSearch, *pHelp;public:
        zString _Edit1, _Edit2, _Edit3, _Edit4, _Edit5;
        zString _Edit7;
        zString _Edit6, _HomeNum, _FaxNum, _SSN;
        int _MaleGroup;
        zString _ComboBox1;
        DAddress(zWindow *, const zResId&);
        int clickedSearch(zNotifyEvt*);
        int clickedHelp(zNotifyEvt*);

        // zpb_begin DAddressClass
        // zpb_end
};

//
// address.cpp - MDI Sample Application
// Version - 1.0
// 5/13/94 - 4:31:30pm
//
//
// zpb_begin ModuleRevisions
// zpb_end

```

```
#include "zapp.hpp"
```

```
#include "tutor.hpp"
```

```
// Main App Class Definitions
```

```
& Includes
```

```
ZAPP_IMPL_ASSERTS;
```

```
// zpb_begin ModuleVars
```

```
// zpb_end
```

```
//
```

```
//      Initial data for list & combo boxes for Address
```

```
//
```

```
char *AddressComboBox1Data[] = {
```

```
    "Employee",
```

```
    "Vendor",
```

```
    0
```

```
};
```

```
//
```

```
// Dialog Member Functions - Address
```

```
//
```

```
DAddress::DAddress(zWindow *w,const zResId& rid) : zFormDialog(w,rid)
```

```
{
```

```

    pEdit1 = ZNEW zEditLine(this, ID_EDIT1, &_Edit1,
FLD_NOTREQUIRED);

    pEdit2 = ZNEW zEditLine(this, ID_EDIT2, &_Edit2,
FLD_NOTREQUIRED);

    pEdit3 = ZNEW zEditLine(this, ID_EDIT3, &_Edit3,
FLD_NOTREQUIRED);

    pEdit4 = ZNEW zEditLine(this, ID_EDIT4, &_Edit4,
FLD_NOTREQUIRED);

    pEdit5 = ZNEW zEditLine(this, ID_EDIT5, &_Edit5,
FLD_NOTREQUIRED);

    pEdit6 = ZNEW zStringEdit(this, ID_EDIT6, &_Edit6, "99999",
FLD_NOTREQUIRED|FLD_SHOWGUIDES);

    pHomeNum = ZNEW zStringEdit(this, ID_HOMENUM,
&_HomeNum, "'(999)' '999'-'9999'",
FLD_NOTREQUIRED|FLD_SHOWGUIDES);

    pFaxNum = ZNEW zStringEdit(this, ID_FAXNUM, &_FaxNum,
"'(999)' '999'-'9999'", FLD_NOTREQUIRED|FLD_SHOWGUIDES);

    pSSN = ZNEW zStringEdit(this, ID_SSN, &_SSN, "999'-'99'-'9999",
FLD_NOTREQUIRED|FLD_SHOWGUIDES);

    pEdit7 = ZNEW zEditBox(this, ID_EDIT7, &_Edit7,
FLD_NOTREQUIRED);

    pComboBox1 = ZNEW zComboBox(this, ID_COMBOBOX1,
&_ComboBox1);

    pComboBox1->addCharStrings(AddressComboBox1Data);

```

```

pComboBox1->selection(0);
_MaleGroup = ID_FEMALE; // Set initial check
ZNEW zRadioGroup(this, ID_MALE, ID_FEMALE, &_MaleGroup);
pSearch = ZNEW zPushButton(this, ID_SEARCH);
pHelp = ZNEW zPushButton(this, ID_HELP);
pSearch->setNotifyClicked(this,
(NotifyProc)(NotifyEvtProc)&DAddress::clickedSearch);
pHelp->setNotifyClicked(this,
(NotifyProc)(NotifyEvtProc)&DAddress::clickedHelp);
sizer()->update();
// zpb_begin DAddressConstructor
// zpb_end
centerWindow(this); // Center window on parent
show();
}

```

```

int DAddress::clickedSearch(zNotifyEvt* ev) {
    DSearch* p=ZNEW DSearch(this, zResId(IDD_Search));
    p->modal();
    if (p->completed()) {
        // zpb_begin DAddressSearch
        // zpb_end
    }
    delete p;
}

```

```

        return 1;
    }

int DAddress::clickedHelp(zNotifyEvt* ev) {
    // zpb_begin DAddressclickedHelp
    // zpb_end
    return 1;
}

// zpb_begin DAddressMemberFunctions
// zpb_end


// zpb_begin AddressUserCode
// zpb_end


//search.hpp
class DSearch : public zFormDialog {
    zListBox *pListBox1;
    zButton *pOK, *pCancel;
public:
    zString _ListBox1;
    DSearch(zWindow *, const zResId&);

```

```

        // zpb_begin DSearchClass
        // zpb_end
};

//
// search.cpp - MDI Sample Application
// Version - 1.0
// 5/13/94 - 4:31:30pm
//
//
// zpb_begin ModuleRevisions
// zpb_end

#include "zapp.hpp"
#include "tutor.hpp"                                // Main App Class Definitions
& Includes

ZAPP_IMPL_ASSERTS;

// zpb_begin ModuleVars
// zpb_end

//
// Dialog Member Functions - Search

```

```

//
DSearch::DSearch(zWindow *w,const zResId& rid) : zFormDialog(w,rid) {
    pListBox1 = ZNEW zListBox(this, ID_LISTBOX1, &_ListBox1,
FLD_NOTREQUIRED);
    sizer()->update();
    // zpb_begin DSearchConstructor
    // zpb_end
    centerWindow(this);    // Center window on parent
    show();
}

// zpb_begin DSearchMemberFunctions
// zpb_end


// zpb_begin SearchUserCode
// zpb_end


//personal.dlg
IDD_Personal DIALOG 6, 17, 209, 175
STYLE
WS_CHILD|WS_BORDER|WS_SYSMENU|WS_MINIMIZEBOX|WS_CA
PTION
CAPTION "Personal Information"

```

FONT 8, "Helv"

BEGIN

```
    CONTROL "Employee:", -1, "Static",
SS_LEFTNOWORDWRAP|WS_CHILD|WS_VISIBLE, 4, 4, 40, 8
    CONTROL "", -1, "Button",
BS_GROUPBOX|WS_CHILD|WS_VISIBLE|WS_GROUP, 4, 12, 200, 48
    CONTROL "First Name:", -1, "Static",
SS_LEFTNOWORDWRAP|WS_CHILD|WS_VISIBLE, 7, 18, 40, 8
    CONTROL "M.I.", -1, "Static",
SS_LEFTNOWORDWRAP|WS_CHILD|WS_VISIBLE, 76, 18, 16, 8
    CONTROL "Last Name:", -1, "Static",
SS_LEFTNOWORDWRAP|WS_CHILD|WS_VISIBLE, 118, 18, 40, 8
    CONTROL "", ID_FIRSTNAME, "Edit",
WS_CHILD|WS_BORDER|WS_VISIBLE|WS_TABSTOP|WS_GROUP|ES
_LEFT, 8, 28, 52, 12
    CONTROL "", ID_MI, "Edit",
WS_CHILD|WS_BORDER|WS_VISIBLE|WS_TABSTOP|ES_LEFT, 76,
28, 16, 12
    CONTROL "", ID_LASTNAME, "Edit",
WS_CHILD|WS_BORDER|WS_VISIBLE|WS_TABSTOP|ES_LEFT, 118,
28, 80, 12
    CONTROL "Social Security:", -1, "Static",
SS_LEFT|WS_CHILD|WS_VISIBLE, 9, 46, 56, 8
```

CONTROL "", ID_SOCSEC, "Edit",
 WS_CHILD|WS_BORDER|WS_VISIBLE|WS_TABSTOP|ES_LEFT, 76,
 44, 76, 12
 CONTROL "Home:", -1, "Button",
 BS_GROUPBOX|WS_CHILD|WS_VISIBLE, 4, 64, 200, 84
 CONTROL "Street Address:", -1, "Static",
 SS_LEFTNOWORDWRAP|WS_CHILD|WS_VISIBLE, 8, 78, 68, 8
 CONTROL "", ID_ADDRESS, "Edit",
 WS_CHILD|WS_BORDER|WS_VISIBLE|WS_TABSTOP|WS_GROUP|ES
 _LEFT, 8, 89, 192, 12
 CONTROL "City:", -1, "Static",
 SS_LEFTNOWORDWRAP|WS_CHILD|WS_VISIBLE, 8, 104, 32, 8
 CONTROL "State:", -1, "Static",
 SS_LEFTNOWORDWRAP|WS_CHILD|WS_VISIBLE, 116, 104, 28, 8
 CONTROL "Zip Code:", -1, "Static",
 SS_LEFTNOWORDWRAP|WS_CHILD|WS_VISIBLE, 148, 104, 32, 8
 CONTROL "", ID_CITY, "Edit",
 WS_CHILD|WS_BORDER|WS_VISIBLE|WS_TABSTOP|ES_LEFT, 8,
 115, 100, 12
 CONTROL "", ID_STATE, "Edit",
 WS_CHILD|WS_BORDER|WS_VISIBLE|WS_TABSTOP|ES_LEFT, 116,
 115, 20, 12

```

CONTROL "", ID_ZIP, "Edit",
WS_CHILD|WS_BORDER|WS_VISIBLE|WS_TABSTOP|ES_LEFT, 148,
115, 52, 12

CONTROL "Telephone:", -1, "Static",
SS_LEFT|NOWORDWRAP|WS_CHILD|WS_VISIBLE|WS_GROUP, 10,
134, 40, 8

CONTROL "", ID_TELEPHONE, "Edit",
WS_CHILD|WS_BORDER|WS_VISIBLE|WS_TABSTOP|ES_LEFT, 56,
132, 80, 12

CONTROL "OK", IDOK, "Button",
WS_CHILD|BS_PUSHBUTTON|WS_VISIBLE|WS_TABSTOP, 48, 156,
36, 14

CONTROL "Cancel", IDCANCEL, "Button",
WS_CHILD|BS_PUSHBUTTON|WS_VISIBLE|WS_TABSTOP, 125, 156,
36, 14

END

```

```
//address.dlg
```

```

IDD_Search DIALOG 37, 13, 161, 146
STYLE WS_POPUP|DS_MODALFRAME|WS_SYSMENU|WS_CAPTION
CAPTION "Address Search"
FONT 8, "Helv"
BEGIN

```

```

CONTROL "", ID_LISTBOX1, "ListBox",
WS_CHILD|WS_VISIBLE|WS_BORDER|LBS_SORT|LBS_NOINTEGRA
LHEIGHT|WS_VSCROLL, 15, 21, 131, 95
CONTROL "OK", IDOK, "Button",
WS_CHILD|BS_DEFPUSHBUTTON|WS_VISIBLE|WS_TABSTOP, 41,
126, 32, 12
CONTROL "Cancel", IDCANCEL, "Button",
WS_CHILD|BS_PUSHBUTTON|WS_VISIBLE|WS_TABSTOP, 80, 126,
32, 12
CONTROL "Names", -1, "Button",
WS_CHILD|BS_GROUPBOX|WS_VISIBLE, 6, 8, 148, 114
END

```