1993

Object-oriented method for graphical user interface design in a distributed system environment

Li Sun Zheng

The University of Montana

Follow this and additional works at: https://scholarworks.umt.edu/etd

Let us know how access to this document benefits you.

Recommended Citation
Zheng, Li Sun, "Object-oriented method for graphical user interface design in a distributed system environment" (1993). Graduate Student Theses, Dissertations, & Professional Papers. 5108.
https://scholarworks.umt.edu/etd/5108

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.
Object-Oriented Method for Graphical User Interface Design in a Distributed System Environment

Li Sun Zheng
B. S., Sichuan University

Presented in partial fulfillment of the requirements for the degree of Master of Science
University of Montana
1993

Approved by

Chairman, Board of Examiners

Dean, Graduate School

Date
May 5, 1993
Zheng, Li Sun, M.S., April 1993

Object-Oriented Method for Graphical User Interface Design in a Distributed System Environment

Director: Professor Raymond Ford

Today's applications tend to be much larger and more sophisticated, and thus more difficult to develop than a decade ago. Their functionality is shifting from processing to system simulation and integration, from text-based to graphics and multimedia based systems, and from centralized to distributed computing.

This thesis presents an object-oriented method for graphical user interface design in a distributed system environment. The method, along with a complete design and implementation, are illustrated in the context of an embedded system development environment.
# Contents

1 Object-Oriented Analysis and Design with an Example .............................. 1  
1.1 Overview of Object-Oriented Techniques .............................................. 1  
1.2 Environment, Requirements, and Implementation Approach ..................... 5  
1.3 Prospective Research and Comparative Studies ..................................... 10  
1.4 Objects Classification and Classes Tree ............................................ 11  
1.5 Download Algorithm – an Example ....................................................... 21  

2 The Graphical User Interface Design of EVBTool and Message-Driven Programming ............................................................... 26  
2.1 Visible Language and Its Primary Components ...................................... 27  
2.2 The Features and Advantages of Graphical User Interface ...................... 28  
2.3 The Consistent Look and Feel of the EVBTool Environment .................... 31  
2.4 Message-Driven Programming ............................................................... 37  
2.5 Message-Driven in OFFLOAD Program .................................................. 42  

3 The Comparative Views of EVBServer in Unix Environment and EVBTool with MS-WINDOWS 3.1 ................................................................. 47  
3.1 CONNECTION Design in EVBServer ...................................................... 48  
3.2 CONNECTION — Design and Communication with Outside World in the Windows Programming Environment ..................................................... 52  

4 Trend, Future Direction, and Conclusion ................................................. 57  

5 BIBLIOGRAPHY ................................................................................. 62
List of Figures

1.1 The development life cycle of an object-oriented software system. 4
1.2 The distributed environment of a software development system
   for the embedded system board EVB/EVBU. 6
1.3 Classes tree for the EVBTool system. 19

2.1 A typical screen with multiple windows, icons, buttons, scroll
   bars, and pull down menu. 30
2.2 The offload window layout in the EVBTool software package. 32
2.3 The message flow diagram for a typical windows program. 41
2.4 The message flow in OFFLOAD. 43

3.1 The flow chart of CONNECTION logic. 49
3.2 The message flow in CONNECTION section. 55
ACKNOWLEDGEMENT

I wish to thank all the faculty, staff and my colleague at the University of Montana for their support and encouragement. My academic advisor Professor Raymond Ford has gone through all the way guiding and helping me for my studies and research; reading, correcting, and refining almost every paragraph of the manuscript. Special thanks also go to Professors James Ullrich, Alden Wright and Keith Yale, who taught me courses, and have always been available when I need guidance, literature, and software.
Chapter 1

Object-Oriented Analysis and Design with an Example

1.1 Overview of Object-Oriented Techniques

Object-Oriented techniques help software designers and programmers to simplify the way they view the real world, and to translate this view through programming into software systems. Superficially, the term "Object-Oriented" means that the software is organized as a collection of discrete objects that incorporate both data structure and behavior. An object is a collection of attributes (data structure) and operations (behavior) designed to emulate a physical or abstract entity. Objects represent a direct abstraction of commonly used items, and they hide most of their implementation complexity from their users. Each object has a unique name, which other objects use to address it. Objects communicate with each other by sending and receiving
messages. A message is addressed to the receiving object and contains the name of a specific operation of the receiving object that is to be executed, along with parameters such as the names of other objects.

Object oriented techniques are founded on a collection of powerful ideas: abstraction, encapsulation, reuse and inheritance.

**Abstraction** consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties.

**Encapsulation** (also information hiding) consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects.

**Reuse** is the act of designing components to be useful in several applications, including applications that are unknown at present. Object-Oriented development systems attempt to build libraries of reusable components for future projects.

**Inheritance** of both data structure and behavior allows common structure to be shared among several similar subclasses without redundancy.

Object-Oriented methods give great importance to relationships between objects, rather than implementation details. Relationships are ties between objects and are usually developed through genealogical trees in which new
object types are developed from others. Hiding the implementation of an object allows the software designer to concentrate on more substantial infrastructures, thus focusing attention on an object's relation to the rest of the system, rather than how an object's behaviors are implemented.

The first step in developing object-oriented software is to convert the informal user requirements for the software system into a descriptive object model through system analysis. The core of the design procedure is the establishment of the class tree (network), which reflects the relationships and hierarchy among classes and objects. The class tree interacts through abstraction and generalization with the library of reusable objects and classes. Figure 1.1 shows the development life cycle of a typical object-oriented software system.

The primary goal of object-oriented analysis is the development of an accurate and complete representation of the problem domain. It tries to identity the type of objects that map into elements of the application domain to be modeled. To do this we must examine requirements, analyze their implications, and restate them rigorously. We must abstract important real-world features first and defer small details until later implementation stages. A successful analysis model states what must be done, without restricting how it is done, and avoids implementation decisions. The result of analysis should be a thorough understanding of the problem, which provides a strong preparation for design.
Figure 1.1: The development life cycle of an object-oriented software system.
Identifying key elements of the problem domain and related objects is the very first step in our software design.

1.2 Environment, Requirements, and Implementation Approach

EVBTool: The software to be developed is an integrated software tool and graphical user interface for an embedded development system, based on the Motorola 68HC11 EVB board. EVBTool is to be developed under Microsoft Windows 3.1 on IBM PC compatible computers, as part of the development toolkit for the embedded system. Target software developed in the EVB toolkit environment will execute on the EVB/EVBU board. During target system development the EVBTool running on the IBM PC or compatible communicates with the EVB through RS232 serial communication line. Working jointly with BUFFALO, monitor software executing on the EVB/EVBU board, the EVBTool under the MS Windows 3.1 forms the core of the distributed system development environment.

Figure 1.2 depicts the major hardware components and the data flow within the distributed system environment. The development system host is an IBM PC compatible 486DX/33MHz machine with 4 MB RAM and 130MB hard disk, running MS-DOS 5.0 and MS-WINDOWS 3.1. The IBM PC compatible is an ideal host, as the most popular and affordable, yet pow-
Figure 1.2: The distributed environment of a software development system for the embedded system board EVB/EVBU.
erful computer system today. The target system is the Motorola M68HC11 family of microprocessors, which is one of the most commonly used microprocessors in use today for embedded system applications. The M68HC11 target is packaged as an EVB/EVBU board, containing a CPU, I/O ports, ROM, EPROM, and RAM. The EVB board has a M68HC11A1 microprocessor while the EVBU board contains the M68HC11E9. Both processors are members of the versatile, low cost M68HC11 microprocessor family from MOTOROLA. The boards themselves are popular in embedded systems because of their very low cost.

The EVB/EVBU board is used as a "prototype" target board to develop the embedded system applications. An embedded computer system is a computer physically embedded in a larger device designed to implement special types of computational control. Embedded system applications are used extensively in both high technology areas and consumer-oriented applications, ranging from computer components, data communications and vehicle control to microwaves, automatic cameras, and telephone answer machines. Because of their very specialized nature, it can be quite difficult and expensive to build an embedded system. Developing an efficient and user-friendly tool that assists in embedded system development is indeed very desirable and useful.

The development environment supplied with the EVB/EVBU is limited to Motorola's BUFFALO monitor, which supports basic forms of EVB in-
teraction and provides operation, testing, debugging, and communication. Additional shareware developed by third parties, including the EVBTool software developed here at the University of Montana, is distributed free of charge by a consortium sponsored by Motorola.

On the PC host, Microsoft Windows is rapidly becoming the most popular graphical user interface environment for tens of millions of microcomputer users. Microsoft Windows is a multitasking, graphics-oriented alternative to the character-based environment provided by MS-DOS on PC compatible systems. More importantly from the graphics programmer's point of view is that Windows provides an object-oriented programming environment with access to graphical objects such as windows, icons, menus, and mouse events through high-level languages such as Borland C++, Turbo Pascal for Windows, and Visual Basic. A user-friendly PC executive information system would be unachievable without a GUI. The bit-mapped displays and mouse-driven menus provided by Microsoft Windows have opened the door to running graphics-oriented software on the PC. Microsoft Windows also provides an easy-to-use icon-based Program Manager for running programs as well as a File Manager and Print Manager for file maintenance and printer-queue management. However, all the advantages come with significant programming complexity. Without appropriate software development tools, Windows programming can be next to impossible.

Object-oriented techniques are well matched to the challenges of graph-
ics oriented programming. Because of the internal structure of Windows is object-based, when programming for Windows the programmer is really engaged in an object-oriented programming exercise. Thus, the object-oriented techniques and Windows programming are a truly natural combination.

Under Windows, a "window" is a rectangular object on the screen. A window receives user input from the keyboard or mouse and displays graphical output on its surface. Dialog boxes are additional windows, which also are objects. Moreover, the surface of a dialog box always contains several additional windows called "child windows", which take the form of push buttons, check boxes, list boxes, and scroll bars. The user sees all these different windows as objects on the screen and interacts directly with these objects by "using mouse actions to push a button" or "scroll a scroll bar".

Not surprisingly, the most natural implementation target for an object-oriented design technique is an object-oriented programming language. It is relatively easy to implement an object-oriented design with an object-oriented programming language, since language constructs are similar to design constructs. For the implementation of EVBTool software, BORLAND C++ 3.1, an object-oriented programming language is chosen.

Borland C++ 3.1 is the latest release of the best-selling object-oriented development compiler by Borland International Inc. for INTEL 80386/80486 based microcomputers. In addition to supporting for the Windows 3.1 API,
Borland C++ aims to offer “all you need to produce Windows applications”. Borland C++ 3.1 has considerably extended the functionality of its Windows-hosted integrated development environment (IDE). The IDE permits the programmer to develop optimized Windows code under Windows, and to debug multiple applications. Borland C++ 3.1’s OWL (ObjectWindows Library) provides a significantly abstracted object-oriented interface to the Windows API. In addition, the widely used *Microsoft Foundation Classes Library* can be used with this new version of Borland C++. The Borland C++ 3.1 package includes the *Resource Workshop*, an integrated tool for specifying the menus, dialog boxes, string constants, and bit maps that form the *look and feel* of an application, and for modifying these resources within an existing application. Borland software’s stability, portfolio of features, and ease of use make it the obvious choice for object-oriented programmer.

In summary, MS-WINDOWS 3.1 is the software platform and environment, and BORLAND C++ 3.1 is the major programming tool for the EVBTool software.

1.3 Prospective Research and Comparative Studies

The author has developed a software system similar to the proposed system in a previous project. The software, called *EVBServer*, was developed on a
Sun workstation as a software package to integrate capabilities under SunOS
(a variation of UNIX) and BUFFALO. EVBServer was implemented based
on traditional design and programming techniques, using the C language.
EVBServer does not include a graphical user interface.

The EVBTool is designed to be more flexible, extensible, and easier to use,
and through its GUI, to have a more attractive appearance than EVBServer.
The next step of research would be to implement a corresponding version of
the EVBTool in the X/Motif environment, and to compare the two different
window programming environments.

Compared to an object-oriented language approach, the programming en­
vironment MOTIF for X/Window, including programming library for GUI is
implemented in non-object-oriented language C. Without an object-oriented
PL interface, MOTIF programmers have to take a substantially more signif­
icant effort to circumvent programming complexity.

1.4 Objects Classification and Classes Tree

A short summary of the major requirements for the EVBTool is given below:

- To offload data from any specified memory area in the EVB board to
  a file in PC file system, and then to allow the file to be displayed and
  be modified in the PC/windows environment.
- To download S-record files from the PC file system to the EVB board memory using the existing S-record LOADER utility on the EVB board but under control of the PC/windows environment. The S-record file is an object file which is generated by PC-compatible Cross Assembler. It encodes program and object modules into a printable (ASCII) format. This allows viewing of the object file with standard tools. The S-record file includes executable code and information of loading address and error checking.

- To provide a terminal emulator that converts a PC window and keyboard into a logical "EVB terminal", to take full advantage of the BUFFALO utilities in the EVB environment.

Starting from these requirements, different types of objects with specific names have been designated. This process gives the designer a better understanding of the problem, clarifying what needs to be done. It also helps the designer grasp the structure of the model, reducing design complexity by reformulating the problem in more concrete terms.

From analysis of the EVBTool requirements, the following initial set of objects have been created.

<table>
<thead>
<tr>
<th>OBJECT</th>
<th>DEFINITION</th>
</tr>
</thead>
</table>

12
ConnectWindow
Create window, close window, resize window
move window, menu bar, title bar, pop menu, communicate with EVB board

OffloadWindow
Create window, close window, resize window
move window, menu bar, pop menu, title bar, offload data from EVB memory
to the PC file system

DownloadWindow
Create window, close window, resize window
move window, menu bar, pop menu, title bar, download S-record file to the EVB/EVBU board

ScrollingWindow
Create window, close window, resize window
move window, menu bar, pop menu, title bar, scroll the contents of the window

TextWindow
Create window, close window, resize window
move window, menu bar, pop menu,
title bar, display a text file, modify a text file, save a text file

DialogWindow
Create window, close window, move window, title bar, display the message to user, get information from user

MessageBox
Create MessageBox, close MessageBox, move MessageBox, display the message to user

AboutBox
Create AboutBox, close AboutBox, move AboutBox, display the information about the application

OKButton
Create OKButton, get user's choice

CancelButton
Create CancelButton, get user's choice

Application
Create a Application, represent a application program, open the Application
ValidFileName  Get a file name from user, check its validity, put the valid file name in a character string

ValidStartAddr  Get an offload start address from user, check its validity, put the valid start address in a character string

ValidEndAddr  Get an offload end address from user, check its validity, put the valid end address in a character string

Object classification is a process of grouping all objects with the same data structure and behavior into a class, which is an abstraction that describes those properties important to an application and ignores any other properties. A class describes a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. Objects in a class have the same attributes and behavior patterns. For example, in the EBVTool design, objects ValidStartAddr and ValidEndAddr can be grouped into a single class called Validation.

From the objects listed above, classes can be defined according to the three basic types of information:
1. The operations that class can provide. An operation is a function or transformation that may be applied to or by objects in a class.

2. The attributes in the class. An attribute is a data value held by the objects in a class. An attribute should be a pure data value, not an object.

3. The public and private attributes or operations.

For example, for the DialogWindow object, there is a class called TFileOpenDialog that contains following structure:

```cpp
CLASS TFileOpenDialog {
    char file_name [80];
    char path [80];

    public:
        getfilename;
        initDialog;
        display_file_names;
        createOKButton;
```
createCancelButton;

There are no private attributes and operations in this class.

After classes are defined, each class is checked to determine whether it is composed of lower level classes or whether any new classes are implied by it. An effort is made to identify possible inheritance, which is a mechanism for sharing the code or behavior common to a collection of classes by factoring shared properties of classes into superclasses.

A class can be defined broadly and then refined into successively finer subclasses. Each subclass incorporates, or inherits, all of the properties of its superclass and adds its own unique properties. The properties of the superclass need not be repeated in each subclass. Inheritance can be added in two directions: by generalizing common aspects of existing classes into a superclass (bottom up) or by refining existing classes into specialized subclasses (top down). Programmers can therefore specify incremental changes of class behavior in subclasses without modifying already specified classes. Subclasses inherit the code of superclasses, to which they can add new operations and possibly new instance variables.
Inheritance plays an important role in modeling because it can express relations among behaviors such as classification, specialization and generalization. The ability to factor out common properties of several classes into a common superclass and to inherit these properties from the superclass can greatly reduce repetition and redundancy within designs and programs. Inheritance is one of the main advantages of an object-oriented software system. Figure 1.3 shown the classes tree, or class hierarchy for the EVBTool software system following the inheritance analysis.

There are twenty-seven classes and subclasses in the system. Inheritance plays an important role in reducing the program size and programming cost.

Analysis is usually defined as a process of extracting and codifying user requirements and establishing an accurate model of the problem domain. Design, by contrast, is the process of mapping requirements to a system implementation that conforms to desired performance and quality parameters. The analysis phase determines what the implementation must do, and the system design phase determines the plan of attack. The object design phase determines the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations.

Object-oriented design transforms the analysis classes into a model that is very close to the target solution space. It starts from the analysis classes
Figure 1.3: Classes tree for the EVBTool system.
and produces additional types of objects that become classes not directly related to the problem space. These classes extend, generalize, or implement the initial set of analysis classes. Object-oriented design constructs detailed internal views of classes, including definitions of their various behaviors. The focus of object design is the data structures and algorithms needed to implemented each class. The object classes from analysis are still meaningful, but they are augmented with computer-domain data structures and algorithms chosen to optimize important performance measures.

One of the goals of object-oriented design technology is to treat classes as “black boxes”, whose external interface is public but whose internal details are hidden from view. Encapsulating and hiding internal information permits the implementation of a class to be changed without requiring any clients of the class to modify code. Encapsulation and information hiding prevent the components in a program from becoming so interdependent that a small change has massive ripple effects. One may want to change the implementation of an object to improve performance, fix a bug, or consolidate code. If changes are made to the object declarations, the programmer must determine their effects on the code and modify the code accordingly. The encapsulation provides and enforces modularity of classes to prevent changes from propagating through the entire program. Thus, the implementation of an object can be changed without affecting the applications that use it. Encapsulation is not unique to object-oriented languages, but the ability to
combine data structure and behavior in a single entity makes encapsulation cleaner and more powerful than in conventional languages that separate data structure and behavior.

In the EVBTool there is a class ‘THSAddrEdit’ that edits hexadecimal start addresses entered by user, and a class ‘THEAddrEdit’ that edits hexadecimal end addresses. Both classes can be modified to edit decimal instead of hexadecimal addresses, allowing the user to type in start and end address using decimal numbers. But the whole EVBTool software stays untouched. The encapsulation gives software developer an extremely easy way to modify and improve existing software system.

1.5 Download Algorithm – an Example

The following is the algorithm for the function *downlead* in the class DownloadWindow. The lines started with // are explanatory comments.

1. Display file list dialogue which is under current directory with OK_button and Cancel_button.

   // The dialogue box is an object. It provides a visual
   // and functional context for presenting options from
   // which the user can select. Any interactive exchange
   // of information between the user and the system that
// takes place in a limited spatial context is
// considered a dialogue object.
// The OK_button and Cancel_button are different special
// objects. They are functional like the function keys
// except that mouse, instead of keyboard, is the active
// component. Moving the cursor by mouse into the
// OK_button and push the left button on the mouse means
// the system will accept the file name that the user
// chooses. Similarly, Cancel_button means a request to
// system ignoring the file name that the user chooses.

2. Get the file name the user wants to download. If the
   file name is on the file list the user just uses the
   mouse to select the file. Otherwise, the file name with
   full path needs to be typed in.

   // A small rectangle area with blinking vertical bar
   // as a prompt is inserted in the dialogue box.
   // The file name chosen from the file list below the
   // prompt box will be displayed in the small prompt
   // box. The file name with full path can be typed
   // from keyboard into the prompt box. The file name
3. Check what the user did. If user pressed OK_button go to step 4. If user pressed Cancel_button, exit to window original state.

4. Open the file for read. If there is an error in opening file, display an error message and go to step 1, otherwise go to step 5.

// The error message is displayed in a message box which is another object. This message box provides critical information to the user. It typically appears only when the system has entered an unrecoverable problem such as 'can not open the file' in our situation.

5. Get the file size S and create a buffer with size S + 1.

6. Read the file and put it in the buffer, and put '\0' at the end of the buffer.
7. Close the file.

9. int i = 0
   while (buffer[i] != '\0')
   {
      write buffer[i] to the communication port;
      if (error in write to the communication port)
         display error message;
      i = i + 1;
   }

10. display 'download is down' message and return to the window original state

After the analysis and design, the implementation of an object-oriented design is easiest using an object-oriented language. Writing code should be straightforward, almost mechanical, because all the difficult decisions should already have been made during design. The code should be a simple translation of the design decisions into the peculiarities of a particular language.

Significant effort is made in the analysis and design phase, which is the most crucial to the final quality of the entire target software system. Whenever inappropriate types of objects or awkward structures are selected, the
completed system reflects these poor decisions architecturally. Fortunately, with the help of abstraction, inheritance and encapsulation, the effort of designer can be more focused to the essential parts of the system.
Chapter 2

The Graphical User Interface Design of EVBTool and Message-Driven Programming

Modern computers enable everyone to layout, edit, display, and publish graphics of all kinds: charts, maps, diagrams, photos, and illustrations. Computer graphics uses more graphic symbolism, which includes various signs, icons, and symbols, and color than ever before and even includes animation in screen display. To produce these computer graphics is one achievement; to communicate the facts, concepts, and emotional values is another. To be successful, computer graphics must communicate visually in an effective manner.

Improving visual communication can make graphical user-interface design more efficient. On the other hand, graphics-oriented programming presents more challenges to the programming task.
2.1 Visible Language and Its Primary Components

A primary technique to achieve improved visual communication is to use clear, distinct, consistent visible language which refers to all the verbal and visual signs that convey meaning to a viewer. For the visible language, the window is a basis for appearance and interaction. The viewer is looking at a flat background, with one or more rectangular windows in front of the background plane. The windows may tile the foreground or may overlap in various ways. Icons, pull down or pop-up menus, dialogue boxes, or other small signs, standing for objects, can appear on the background plane or in the window planes.

The use of icons, signs, and symbols is popular in computer graphics because icons can be entertaining, clever, and visually appealing, and an individual sign may take up fewer pixels and less space than the equivalent in words. Therefore, more information can be packed into a given window or screen space. The space savings in menus, maps, and diagrams can be significant. Even more important, icons and symbols can replace written languages and contribute to interfaces that are international in design and comprehension. With just a few pixels, an entire situation and course of action can be implied.
2.2 The Features and Advantages of Graphical User Interface

There are many commercial graphical user-interface (GUI) products available today. Good examples are the Apple Macintosh OS, OPEN LOOK GUI, Microsoft Windows, and OSF/Motif Window Manager and Toolkit. Those are all windowing systems and are all based on a common user interface concept. A common user interface concept is a set of rules used across a group of products that formally specifies the visual presentation of data and functions, the user’s interaction, and the logical organization and behavior of information. The visual and functional consistency have several advantages: when the interface for applications is consistent, users can move from one application to another with ease and speed. Consistency facilitates the learning process and minimizes the need for training when new applications are introduced into the workplace, resulting in increased productivity. The users become familiar with simple, clear, and consistent functions and features quickly. Consistency also alleviates the confusion introduced by applications with divergent interfaces and eliminates the associated costs in efficiency and training.

Windows are the fundamental interface objects through which data, commands and controls are organized and presented to the user. The windowing mechanism is similar to an operating system in terms of two major functions
interface and management. Instead of file systems or CPU cycles, however, the windowing system manages resources such as screen space and input devices. In GUIs, the windowing system acts as a front end to the operating system by shielding the end-user from the abstract and often confusing syntax and vocabulary of a keyboard-oriented command language. The GUI makes use of graphics on a bitmapped video display. With a GUI, the video display itself becomes a source of user input. It shows various graphical objects in the form of icons and input devices such as buttons and scroll bars. The user can interact directly with these objects by pushing a button or scrolling a scroll bar. The window receives this user input in the form of "messages" to the window. The interactive and graphical environment makes the computers much more friendly. Users no longer need a long period of study to master the skill necessary for his/her applications. On the other hand, several windows programs can be displayed and running at the same time under the GUI environment. The multitasking nature a graphical user interface possesses would greatly increase efficiency and reduce the development period for a complex project. Figure 2.1 is an example screen with multiple windows, icons, buttons, scroll bars, and pull down menu.

A window also utilizes messages to communicate with other windows. This makes a GUI an ideal vehicle for distributed system development environment. The users can benefit from the multitasking and interactive advantages.
Figure 2.1: A typical screen with multiple windows, icons, buttons, scroll bars, and pull down menu.
Unfortunately, most of today’s distributed system development environment, the distributed embedded system development environment in particular, is still text-oriented. This undesirable situation can be remarkably improved through a multitasking graphical user interface development tools.

The EVBTool project is an attempt in filling the gap. The goal of EVBTool itself is to provide a graphical user interface in a distributed environment to develop embedded software through the MC68HC11 board and IBM PC or compatible.

2.3 The Consistent Look and Feel of the EVBTool Environment

All Microsoft Windows programs, including the EVBTool, have the same fundamental look and feel. Even though there are hundreds of windows applications currently available, once you are familiar with the operations of one windows program, you are basically in a good position to easily learn another.

Like other windows applications, for example, the EVBTool offload window has a border region controlled by the windowing system and a content region that the application can freely modify (See Figure 2.2). There is a title bar Offload Data from EVB to PC for the offload window. It gives instructions to user as to what the application window does.
Figure 2.2: The offload window layout in the EVBTool software package
The window can be moved and resized by the user, and the application window can become an icon. Because the purpose of any window is to let the user enter data or to let the application display information, a window starts its life cycle when the application has a need for input or output. A window continues its life cycle until there is no longer a need for it, or the application is terminated. Some windows, such as the window used for the application's main user interface, last the life of the application. Other windows, such as a window used as a dialog box, may last only a few seconds.

Menus are the principal means of presenting options to a user from within an application. Menus let the user issue commands without having to remember the specific names and syntax of the particular command required. A menu is a list of items from which the user can choose; each item represents a command. All applications that have commands should provide menus to give the user access to the commands. The window system uses the select-then-operate command paradigm, which designates that the user first selects the object or objects to which the operation is to apply, then chooses the appropriate command from the menu. The user interface is grounded in a relatively small set of fundamental input elements that define the user's interaction with the computer. The two basic elements are the mouse and the keyboard. User uses mouse or keyboard to select object and command which applies on the object.

There are a total of three menus in the offload window. The first menu
is the "File" menu. Under the "File" menu, there are three operations: Show, Save As, and Exit. If "Show" is selected by user using the mouse, a query box will be displayed because the "Show" command requires additional information from the user before it can be completed. The query box is initiated by the application, not the user. It always presents a predefined model context in which one of the available options must be explicitly selected by the user before any further operations can take place. A query box allows the system to request a specific piece of information from the user. In this query box with the "Show" command, there is a list of files in the current directory and two buttons 'OK' and 'Cancel' (See Fig. 2.2).

Buttons are graphical controls that initiate actions and change data object properties or the interface itself. Button controls are the principal interface of a dialog box. Almost all dialog boxes have at least one push-button control and most have one default push button and one or more other push buttons. Many dialog boxes have collections of radio buttons enclosed in group boxes, or lists of check boxes. Users can choose buttons by clicking them with the mouse. Also users can choose any one of the files in the file list by mouse action. If a file the user wanted is not in the current directory, user can type in the complete path. After the file is determined, the user can push the 'OK' button to show the contents of the file or push the 'cancel' button to cancel the "Show" command. The contents are showed in a text window and can be modified by the user. With this window, the horizontal and vertical scroll
bars are on the right and lower sides of it. They let a user scroll through the contents of the client area.

If the “Save As” command is selected, a query box will be displayed. It prompts the user to type in a file name used to save the current information. The content to be saved is displayed in the text window by “Show” command.

The “Exit” command is used to exit from window. It terminates the application program and returns to the basic window interface.

The second menu in the EVBTool software is the “Offload” menu. Only one operation is involved with this menu. That is the “User Info...” operation. A query box is displayed when the “User Info...” command is selected. It asks the user to type in the file name to be used to save the offloaded data from EVB board, the start memory address and the end memory address for the offloading operation. After all the information is typed in, the user can push the “OK” button to issue the “offload” command or push the cancel button to cancel the “offload” command. The system will check the information the user typed in, and it will ask the user to type it again if there is undesirable input.

The third menu is the “Help”. Under this menu, “About” is the only command. A message box will appear when the user chooses the “About” command. It tells the user about the application, such as the version, author, completion date etc. A message box, also called message dialog, provides spe-
specific (critical) information or general information. The critical information is not requested by the user and typically appears only when the system has entered, or is about to enter, an unrecoverable and potentially dangerous state. Often the user is allowed only to acknowledge the receipt of the message before the system attempts to deal with the problem. Alternatively, the user may be also to request more general information. The message box requires a response by the user before work with the application can proceed. The “About” command is an example of this case. It displays information about the application. After the user reads the information, then he or she pushes the “OK” button to continue.

Microsoft Windows allows applications, running simultaneously on the system, to share hardware resources. It is a non-preemptive multitasking environment and runs in INTEL 386/486 processor’s protected mode. The MS-DOS sessions end up running in “almost” real mode while the rest of the operating system, like Microsoft Windows or 386BSD, runs in protected mode in a completely different part of the system’s memory. Microsoft Windows gives Windows applications access to far beyond the 640KB memory limit imposed by DOS. Also a program can contain more code than can fit into memory at any one time. The user can run several instances of the same program, and have all these instances share the same code in memory. Microsoft Windows can discard code from memory and later reload the code from the program’s .exe file. For example, the user can open three
windows in the EVBTool; a download, an offload, and a connect window. In the download window, user can download the S-record file from PC to EVB board. Then in the connect window, the user can execute the downloaded file on the EVB board under BUFFALO. Also, in the offload window the user can offload the data (or results) from EVB board to PC as a file and use the "Show" command to display the data. At the same time, in the connect window, the user can use the 'dm' command in the BUFFALO to display the exact data on the EVB board. Then the user can very easily make a comparison between those two windows to see whether the offloaded data is correct. All the tasks are completed without leaving the Windows/EVBTool environment.

2.4 Message-Driven Programming

The big difference between programming in DOS, or in an environment presented by a traditional character-oriented terminal connected to a minicomputer, and programming in the Windows environment is that instead of telling the computer what to do one step at a time, Windows programs are structured to wait there until the program receives a message from Windows. This is called Message-Driven Architecture. The programming in DOS is linear programming. It has a definite staring point, and follows a definite course to its finish before it terminates. This is the essence of the linear programming model: start at the beginning, run to the end, then stop.
Of course, the linear-model programs can branch, loop and engage in some general jumping around in the code, but their basic nature is to follow a fixed course. Most importantly, there is a definite beginning, a definite end and a bunch of code in the middle, written with the intention that every bit of it executes every time the program runs.

Comparing this familiar operation of a program with the world of message-driven software, everything changes. In message-driven programming, after a program is loaded into memory, there is no guarantee of what comes next. There is no predestined route to the end. Instead, there is no predetermined end at all, no restriction on whether parts of the program do not execute, and no rules requiring one part of the program to run before another. The beauty of message-driven programming is that users still have their work done. They simply get it done in a more comfortable, interactive way. Message-driven programming mirrors the way people think and act.

Messages are statements like “The user just clicked a button with the mouse pointer — do something!” All Windows messages received are ultimately sent to the program’s message processing functions. These messages control the operation of the program. For example, when a user resizes a window, Windows sends a message to the program indicating the new window size. The program can then adjust the contents of its window to reflect the new size. You can think of a Windows program as an obedient slave. The program just sits there waiting for a message. When it receives a mes-
sage, the program does some task and then goes back to just sitting there.

In an message-driven program, the computer must respond to its user's requests and needs. The program can do so because its subroutines are much more like individual strategies. Each software subroutine represents one action. By drawing on this library of pre-written "policies and procedures," the message-driven program can rise to almost any occasion, even responding to unanticipated sequences of events. The message-driven model focuses attention exclusively on the user. The model insists that programmers respond to and adjust to the whims of the user. The user is put in control of program flow. This is completely different from a DOS program. DOS programs have an active "mentality". "First I will do this, then I will do that", etc. There is no real equivalent to a message in the DOS world. Message processing is the basic concept behind Window and most other GUIs. In addition, message processing implies the ability to run several applications at the same time, and allow the user to easily switch between them. Windows programs must be designed to give up control frequently so that other programs have a chance to operate.

Messages are not like interrupts. Windows will not jump in and halt the execution of a program to go to some more critical task. For example, if a calculation is still running after a menu item is clicked, you do not have to worry about the calculation being stopped half way through because a new message is received. Windows will not be able to pass another message to
your program until the calculation is done and the execution returns to the message loop for another message from the message queue. One exception that will start a message from within your function’s calculation is if the program uses a Windows function that sends a message, bypassing the message queue.

Figure 2.3 shows a simplified diagram for a message being processed by a Windows program. The Windows loads low-level functions for dealing with the keyboard, mouse, and screen when the Windows starts. When a hardware event such as a keystroke occurs, Windows sends a message to the active program’s message queue. Naturally, messages do not all start from hardware actions. Programs may find it convenient to send a software message from within the body of the program. This is frequently done in place of using a “goto:” statement. In addition, messages can be sent directly to the \textit{WndProc()} function by \textit{SendMessage()} function. The application queue is first-in/first-out queue and just a memory location to hold message data that has not yet been processed by the running program. A application program pulls messages off the message queue in the “message loop”. The message loop typically has the form

\begin{verbatim}
while (GetMessage(&msg))
{
    TranslateMessage (&msg);
    DispatchMessage (&msg);
\}
\end{verbatim}
The message's action is not defined in the WndProc().

Windows send a message to the Active program's message queue.

The Program's Message Loop:
- GetMessage()
- TranslateMessage()
- DispatchMessage()

The Program's Logic and actions:
- WndProc() Function

If the message's action is not defined in the WndProc(),
- Windows send a message to the program's message queue.
- The Program's Message Loop.
- DefWindowProc() Function

Hardware Event Occurs (for example, pressed keyboard or mouse button).

Other Applications:
- SendMessage()
- PostMessage()

Other Applications:
- SendMessage()
- PostMessage()

Figure 2.3: The message flow diagram for a typical windows program

41
The `GetMessage()` function fetches the message and can give control to other programs if there are no messages to process. `GetMessage()` is followed by `TranslateMessage()` and `DispatchMessage()`. `DispatchMessage()` sends the message data to the `WndProc()` function, which the programmer writes to handle the program’s logic. `DispatchMessage()` knows which function should receive the message because the programmer defined the message function in the Windows’ class definition. `TranslateMessage()` is a utility function that takes raw input from the keyboard and generates a new message that is placed on the message queue. The new message contains the ASCII value corresponding to the key pressed, which is easier to deal with than the raw keyboard scan codes and virtual key codes.

2.5 Message-Driven in OFFLOAD Program

One of the major functions in OFFLOAD is offloading data from EVB/EVBU board to PC and saving the data as a file in the PC file system. Figure 2.4 illustrates the message flow in OFFLOAD.

When the user presses the mouse button to choose “offload” command, a hardware event occurs. Windows sends the “WM_COMMAND” message with associated information to OFFLOAD’s message queue. Once `GetMessage()` in the message loop gets this message, `DispatchMessage()` sends it to
Hardware Event Occurs
Message Queue
PostMessage()
TranslateMessage()
Other programs
SendMessage()

Message Loop:
GetMessage(&msg)
TranslateMessage(&msg)
DispatchMessage(&msg)

WndProc(message, wparam)
{
    switch(message)
    {
        case WM_CREATE:
        case WM_COMMAND:
            switch(wparam)
            {
                case Offload:
                case About:
                case Show:
                case SaveAs:
                    break;
            }
        case WM_DESTROY:
            return;
    }
}

Figure 2.4: The message flow in OFFLOAD
In \textit{WndProc()}, the "WM.COMMAND" message is first switched to the case \textit{WM.COMMAND}, then switched to case \textit{Offload} according to the message information "wparam". In case \textit{Offload}, the command Offload is actually executed. Steps in \textit{Offload} command are following:

1. Get information from user
   - Ask for a file name used to save data
   - Ask for start and end memory addresses for offloading
2. Check all the information user typed in, if there is a error, go back to step 1, otherwise continue
3. If user push "OK", continue
   - If user push "Cancel", go step 9
4. Open the file for write, if there is an error, display error message and go step 9, otherwise continue
5. Prompt user to push reset button on the EVB/EVBU board to get initial state of the board
6. Send "md XX YY", the Buffalo command, to EVB/EVBU board using \texttt{writecom()} function
   - XX is start memory address
   - YY is end memory address
7. Read data sent by EVB/EVBU from the receive data queue to a buffer
8. Write the contents of the buffer to the file which is opened in step 4
9. Return back to message loop to get next available message

Other commands, like Show, SaveAs etc., follow the same routine as the Offload command, except that they use different subprograms in WndProc() to execute different command sequences.

The Offload program only uses the most basic message function: a message loop in WinMain() and a series of actions based on received messages in the WndProc() function. Significantly more control over message processing is possible. Separate programs can communicate with each other by exchanging messages as they progress.

Messages can also be intercepted and modified by hook functions, prior to being passed to the program. Hooks allow a module (running program) to intercept messages going to other applications. The messages can be acted on, modified, or even stopped. A typical example of uses for a hook function would be to remap the keyboard. Every keyboard message could be intercepted, and then modified to reflect a different keyboard layout. More sophisticated uses are to modify the behavior of specific applications. For example, programmer could write a hook function that intercepts WM_SIZE and WM_MOVE messages for a program, forcing the application to be always
located at one spot with the fixed size on the screen.
Chapter 3

The Comparative Views of EVBServer in Unix Environment and EVBTool with MS-WINDOWS 3.1

EVBServer and EVBTool are different versions of the software for developing embedded system software. They are designed for different environments with different user interfaces, using different methodologies. The older EVBServer uses a traditional programming design technique and has a text-oriented user interface; the EVBTool uses an object-oriented programming design technique and has a graphical user interface environment. Another difference is that EVBServer uses the linear programming method and EVBTool uses message-driven programming techniques. The CONNECTION function in the softwares is a good example to illustrate the different design and implementation techniques.
The essence of CONNECTION is to display data from EVB/EVBU on the screen and to send data from the keyboard to the EVB/EVBU through a communication port. Figure 3.1 shows the flow chart of CONNECTION.

3.1 CONNECTION Design in EVBServer

The CONNECTION procedure constantly checks if there is incoming data. If so, it takes it in and puts it on the screen. In the mean time, it watches the keyboard. If a key is hit, it sends it to EVB/EVBU through the serial port.

In the conventional Unix environment, the first thing CONNECTION does is to open the communication port `open ("/dev/ttya", O_RDWR)`, where "/dev/ttya" is the physical device representing communication port and O_RDWR is for setting reading and writing options. The second thing it does is to initialize the status of ttya, the communication port and the procedure’s standard input device, stdin or the keyboard. Both ttya and stdin are initialized to be RAW MODE and 9600 baud rate speed, which is required by the EVB/EVBU board by call `ioctl()`. After that, there is a `while` loop to take care of all the incoming and outgoing data. The loop looks like this:

```c
while(1) {
    ioctl (ln, FIONREAD, &rbytes);
    /* check how many bytes are */
```

Figure 3.1: The flow chart of CONNECTION logic
if (rbytes > 0) { /* available in ttya */
    read (In, buf, rbytes); /* read them into buffer */
    write (1, buf, rbytes);
    /* and display them on the screen */
}
else {
    ioctl (0, FIONREAD, &wbytes);
    /* check if keyboard is hit */
    if (wbytes > 0) {
        read (0, &c, 1) /* if so, read it in */
        if (c == 'q')
            break;
        else
            write (ln, &c, 1); /* send it to the ttya */
    }
}

This procedure will last forever until an 'q' (quit) input is detected.

Another way to do the communication in a Unix environment can be found in the popular data communication program CKERMIT. The CONNECTION module in CKERMIT uses a fork system call to generate a new process. The parent process will take care of data from the keyboard, while
the new process manages ttya, the serial communication port.

Usually, a system consists of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes can potentially execute concurrently. Switching the CPU from one process to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch. When context switch occurs, the state information of the computer's processor (registers, mode, memory mapping information, coprocessor registers, and so on) or context, is saved away in a location where it can be later restored. Then, a new process which must be run is determined. Finally, the state information of the new process must be loaded and run. Multitasking programming suffers an enormous context-switch time penalty. Context-switch time is pure overhead. The kernel of the Unix makes context switches between these two processes, among others, constantly in the CONNECTION module in CKERMIT. The overhead caused by context switching lowers performance compared to the same job being accomplished by a single-task process.

The CONNECTION in the EVBServer is integrated in a single process, which eliminates the overhead for context switch. Therefore it should be a more efficient approach than the way CKERMIT is implemented. But the best way to improve performance is to use the event-driven multitasking technique. In this case, the CONNECTION includes processes that can not
complete without communicating with an external entity. This communication does not require the CPU’s total processing bandwidth. By using an interrupt dedicated to the specific entity to grab CPU time, we not only tell the CPU when to work but also what to work on.

3.2 CONNECTION — Design and Communication with Outside World in the Windows Programming Environment

Implementation of the CONNECTION in the EVBTool is a totally different scenario. Communication support is not a trivial matter for Windows programming. Consider the case of the CONNECTION that sends and receives data through the computer’s serial port. The data from the EVB/EVBU board arrives slowly (relative to the internal clock speed of the computer) and can arrive at any time. If the Windows program simply loops, checking for incoming data bytes, the program would take over the Windows environment. No other application could get the input focus. This violates the basic principle behind the structure of all Windows programs, which must give up control of the environment frequently to allow other programs to run.

To get around this problem, the Windows function library includes interrupt-driven communication support. When the communication port receives an input byte, it generates a hardware interrupt. The interrupt briefly halts
whatever application is running and stores the input byte in a memory buffer. Control is then immediately given back to Windows. The memory buffer is called receive data queue. The data bytes accumulate in the queue as they are received. When the CONNECTION wants to read the incoming data, it reads the receive data queue. Data to be sent to the EVB/EVBU is also stored in a buffer before being sent to the EVB/EVBU. This buffer is called transmit data queue. The size of receive and transmit data queues can be defined by programmer.

There are two approaches to checking the data queues from within a Windows program. The simplest way is to use a timer. The Windows timer can be set to generate a message on a frequency of, say, ten times per second. Each time the message is processed by the application, the receive data queue can be checked with a read function to look for data.

Although the timer approach will work, it is not the best way to design a communication program due to its inflexibility and inefficiency. A better way to write a communication program is to use a PeekMessage() loop for the program’s main message loop. PeekMessage() takes control if no other application is requesting it. This approach allows the communications program to continually check the receive data queue in the “gaps” when other Windows applications are not active.

The PeekMessage() is similar to GetMessage() but more passive. PeekMes-
sage() gives up control as soon as it is called, and it gets control back only if other applications run out of messages to process. A message loop that uses PeekMessage() rather than GetMessage() essentially says to Windows, “Let other programs run for a little while, but once they have emptied their message queues, return control to me. I am not finished with my work”. If two or more programs are running that use a PeekMessage() loop to retrieve messages, Windows uses a round-robin approach, returning control sequentially to each program waiting with a PeekMessage() call (Programming Windows, Petzold [4]). GetMessage() is the most active. It keeps control until it empties out the application program’s message queue. CONNECTION in EVBTool is an typical example of the message-driven structure using the PeekMessage() as the main message loop. Figure 3.2 illustrates the message flow in CONNECTION.

The CONNECTION program continually checks the receive data queue when other applications are not active and no message is in CONNECTION’s message queue. If there are data available in the receive data queue, it reads all them and displays them in the CONNECTION’s window. When user presses any key at anytime, Windows sends a message ‘WM-CHAR’ to CONNECTION’s message queue. Then this message is processed by WinProc(). The characters typed by user are sent to EVB/EVBU through the communication port without display. Thus, whatever the user sees on the connection window is actually coming from EVB/EVBU board, which is often the echo
Figure 3.2: The message flow in CONNECTION section

```c
int Status=WriteComm();
if (Status<0)
    MessageBox("Output Error");
    break;
```

```c
WndProc()
{
    switch(Message)
    {
        case WM_CREATE:
            break;
        case WM_PAINT:
            break;
        case WM_CHAR:
            Status=WriteComm();
            if (Status<0)
                MessageBox("Output Error");
                break;
    }
}
```
generated by the board.

The X-Window based GUIs, such as OSF/MOTIF and OPEN LOOK, create a new species of user interface and programming environment/techniques. This GUIs work in a more flexible way than MSWindows and Macintosh GUI and allow all programs, GUI and non-GUI software, co-exist.
Chapter 4

Trend, Future Direction, and Conclusion

Object-Oriented technology and Graphical User Interfaces has become a phenomenal industry trend. The power of the visual paradigm is now widely recognized and is the foundation for graphics-based systems such as Apple’s Macintosh, Microsoft’s Windows, and MIT’s X Window System development environment for all the operating system environment. Confirmation of the significance of the graphical paradigm is evident by the speed and unanimity with which computer companies have adopted the window/icon/menu/pointer interface. While “user-friendly” graphical interfaces do not necessarily guarantee quality in application programs, one of the most obvious and universal hallmarks of modern, high-quality software is an elegant graphical interface. More and more tools and programming language are available today for software developers to develop graphical user interface ap-
lications, most of which follow the object-oriented programming approach. **Actor** for the MS-Windows platform and **GUIDE** for the OPEN LOOK environment, among others, are just two examples.

Actor developed at the Whitewarer Group, is a full-featured, modern programming environment that is an easier and faster way to produce advanced applications for personal computers than Borland C++. It runs in Microsoft Windows 3.1. Actor's uniformly object-oriented environment allows it to mask the complexity of the window manager without sacrificing extensibility. As a result, the developer can work at a much higher level and let Actor's predefined classes handle the bulk of direct interaction with the windowing platform. At Actor's heart is its object-oriented programming language, which consistently takes advantage of the benefits of inheritance and message sending. In Actor, programmer can develop program elements piece by piece: you can develop and test the windows you need, then create and test functions, then add and test menus and input dialogs, and so on. Programmers can very easily create windows, dialog boxes, menus, and buttons, using a pointing device without writing source code. The code for individual methods in Actor is usually focused and brief. The key to success here in the object-oriented design is figuring out where in the class hierarchy to place new classes, and how to effectively distribute responsibility among objects. Intelligent placement of classes and careful distribution of methods (routines that are performed on objects or classes) lend portability to your
code.

On the other hand, GUIDE, operating in the OpenWindows application environment, a graphical user interface environment that runs on Sun workstations is a development tool designed to make the interface programmer's job easier. GUIDE is an acronym for Graphical User Interface Design Editor — an accurate description of its function. Using it gives you the freedom to create and try user interfaces without ever writing a line of code. You can spend more time designing and testing interfaces, the surest route to better user interfaces for software users.

GUIDE can be used to assemble the elements of a user interface by dragging visual representations of the elements onto the workspace of the monitor screen and putting the elements together. The GUI designer can assemble windows, control areas, panes, buttons, menus, sliders, and other OPEN LOOK UI elements in the interface and then try their operation together without leaving GUIDE.

The user interfaces conform to the OPEN LOOK User Interface as it is implemented in OpenWindows. When you finish assembling the elements of a user interface, you save the interface as an interface file which you can recall later for modification or use to generate source code. GUIDE's companion program, GXV, does the work of coding by reading a GUIDE interface file and generating the C source code necessary to create that interface in Open-
Windows. GXV even generates a Makefile, header file, and stubs file to help the application programmer link the interface code to the main body of the application code. GUIDE is obviously a powerful programmer's tool, but it is also a useful tool for non-programmers such as user interface designers because you can easily create and modify user interfaces without ever generating source code. Since a GUIDE interface does not require days and weeks of code writing, it is much less time-consuming to change the interface. Once you are finished with the interface design, the programmer can use GXV to generate interface source code and then link the interface to the application.

GUI applications are inherently more difficult to develop, and hence to port, than are text-based application. This is one of the reason that cross-platform development has grown more complex in recent years. As a consequence, toolkits to aid in development of cross-GUI applications are emerging to the market. These toolkits are analogous to the cross-platform compilers and assemblers that were the only cross-development tools decades ago.

One of such toolkits, the extensible Virtual Toolket (XVT Software, Boulder, CO) consisting of C libraries for Windows, X11/Motif, X11/Open Look, Presentation Manager (OS/2), Macintosh, and character-based interfaces, which let the user write an application once and compile it on the different target platforms. Rather than provide a complete replacement for native toolkits, XVT's library translates application calls into the form required by
the native toolkit. The result is that applications written using XVT have
the look and feel of applications written directly to the native environment.

In addition to basic function calls, which XVT translated into similar functions in the underlying window system. XVT includes higher level functions. These provide advanced functionality and pull together disparate programming elements.

It is said, every five to seven years some piece of hardware or software comes along that causes the computer industry to change, in a fundamental way, the manner in which people use computers. Without question, graphical user interfaces and object-oriented programming are the software technology for the 1990s.
BIBLIOGRAPHY


