1998

A Java Implementation of a Portable Desktop Manager

Scott J. Griswold
University of North Florida

A JAVA IMPLEMENTATION OF A PORTABLE DESKTOP MANAGER

by

Scott J. Griswold

A thesis submitted to the
Department of Computer and Information Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

April, 1998

The thesis "A Java Implementation of a Portable Desktop Manager" submitted by Scott J. Griswold in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

4/20/98

Dr. Ralph Butler
Thesis Advisor and Committee Chairperson

Signature Deleted

4/20/98

Dr. Yap S. Chua

Signature Deleted

4/20/98

Dr. Judith L. Solano

Accepted for the Department of Computer and Information Sciences.

Signature Deleted

4/21/98

Dr. Charles N. Winton
Chairperson of the Department

Accepted for the College of Computing Sciences and Engineering:

Signature Deleted

4/21/98

Dr. Charles N. Winton
Acting Dean of the College

Accepted for the University:

Signature Deleted

4/22/98

Dr. William J. Wilson
Dean of Graduate Studies

ACKNOWLEDGEMENT

I thank Dr. Ralph M. Butler, my thesis advisor, for his advice and guidance during the development of this thesis. His knowledge and encouragement were greatly appreciated. I am deeply grateful for the advice and contributions of Dr. Yap S. Chua and Dr. Judith L. Solano, the thesis committee members. I also thank my wife, Janet, for her encouragement and support.

CONTENTS

FIGURES

ABSTRACT

Computers equipped with a Graphical User Interface (GUI) and connected to the Internet are common in both the business and educational communities. These computers, using a web browser, easily share programs written in the Java programming language that are able to execute on a variety of heterogeneous machines.

While programs written in many languages will not execute unmodified on different computing platforms because of portability restrictions, Java has overcome these limitations and provides a platform independent language. However, every programming language is limited, and no language provides all the features required for every program. Therefore, creation of any portable program must consider not only the language, but also the architecture and operating system constraints of the target machines. A desktop manager program was developed using the Java programming language. This program provides a uniform user interface to manage other programs and executes on multiple computer platforms.

Chapter 1

INTRODUCTION

Portability of software involves many issues such as machine resources, operating systems, software languages, and the functionality of the software being ported. Problems of software portability were investigated by developing a desktop manager and several desktop utility programs that present a consistent user interface across multiple platforms. These programs were written using the Java programming language that is capable of running on multiple environments without code modification.

This thesis document consists of five chapters. Chapter 1 is an introduction to the problem and the thesis' objective. Chapter 2 details the research on portability and different desktop manager programs. Chapter 3 provides an analysis of the design requirements for a Desktop Manager written in the Java language and the resolution of the portability impediments. Chapter 4 looks at the Java language features that support the program design requirements. Chapter 5 concludes by reviewing the results of the program, areas of success and limitation, and recommendations for future areas of research.

1.1 Problem Review

Portability of software has long been an area of study in computing science. Portability issues gain increasing attention as more computers become connected to networks and especially the Internet. While heterogeneous computers can communicate because they use a common protocol, they are often unable to share programs thus limiting their exchanges solely to data transfer. A notable exception is the Java programming language, developed by Sun Microsystems, Inc., that has demonstrated cross platform portability. One common use for Java allows heterogeneous computers, using a web browser and the HTTP protocol, to transfer Java Applet programs to a client computer where they run locally. Another example of Java's platform independence is shown by Corel Corporation's conversion of their Office Suite programs (WordPerfect, Quattro Pro) to the Java language [Corel97]. This allows their customers to purchase a single copy of the program that will operate on multiple platforms. Previously each program was dependent on the operating system which had to be specified at the time the program was purchased.

Portability is desirable for programs that have a broad market because it allows them to operate on many different platforms. Programs such as text processors, library catalog systems, and payroll processing are used on many different

computers. Often these programs have to be modified or rewritten in another language for them to operate on a different computer. This effort does not enhance the performance or capabilities of the program, but just overcomes the restrictions to portability. "At its most basic level, portability is an economic issue" [Rowley96, page 80] since portability increases the potential market for the software by removing platform restrictions. Developing a program in a portable language expands its operating domain and therefore increases its value.

A desktop manager program provides a Graphical User Interface (GUI) that allows users to control and manage other application and utility programs. Operating systems like Macintosh's and Microsoft's Windows 95 offer desktops as part of their operating system. On systems like UNIX, desktops are available as a separate program. Consequently, none of these have the same look or operation and each is tied to the underlying operating system and not easily ported.

A Desktop manager program written in Java would benefit from portability because it could execute on multiple platforms providing a consistent interface to the user. However, the software language is just one of the impediments to porting a program. Other obstacles are the machine architecture and operating system of the host machine. Therefore, developing

a desktop manager in Java requires the identification and resolution of all limitations that restrict portability.

1.2 Objective

The issues of software portability are investigated by using the Java programming language to develop a desktop manager and several desktop utilities. These programs present a consistent computing interface to the user while being capable of running on the multiple platforms without modification.

Chapter 2

BACKGROUND RESEARCH

2.1 Portability

Olivier Lecarme defines software portability as a quantitative measure of the effort to modify a software program to make it operate on another platform [Lecarme89, page 10]. The platform or environment is the combination of hardware and software (operating system). According to this definition, a software program is 100% portable if no effort is required to port it and 0% portable if the effort to port it is equal to the effort to originally develop the software. Since portability is a quantitative measurement, degrees of portability exist.

Reusable and interchangeable software have been called the silver bullet that will stop the increasing costs of software development [Cox90, page 210]. Also, Lecarme states that "Computer hardware is only as useful as its available software" [Lecarme89, page 1]. Therefore, portable software should improve the capability of a computer and reduce the cost of software development.

There are four major areas that may pose impediments to porting software [Lecarme89, pp. 11-17]. One area is the computer or processor. Processor speed and the number of processors vary on different machines. Additionally, the machine representation of data and memory organization is not uniform for all processors. Programs cannot be ported to machines that do not provide the minimum necessary processor capabilities. This is evidenced by software vendors who provide minimum machine and operating system requirements for their products. However, sometimes the processor can be hidden by software, as is the case with the UNIX operating system that runs on a variety of different processors. In these cases the software often just needs minor modifications for it to run on a different machine.

The machine hardware peripherals like printers, graphic monitors, tape drives, disks, and networks that are used by the software are not the same on all environments. Clearly, if those do not exist or operate in a different manner, the software needs to change. A program that scans data through a scanner would not port to a machine that does not support the use of a scanner. Often device drivers are used to hide the implementation details of peripherals. Then, if devices are changed or new devices installed, only a new software driver has to be written.

Another consideration of porting software is the operating system, itself a software language with semantics and syntax, that controls the hardware resources [Lacarme, page 14]. This creates two distinct problems: porting the operating system to a different machine and porting the applications to a different operating system. Porting the operating system is often a difficult task, but systems such as UNIX have been ported to multiple hardware platforms [Friesenhahn97]. Having the same operating system that can run on many different hardware platforms increases the use of application programs. Porting of application software to different UNIX platforms is easier than porting from UNIX to a different operating system.

All computer programs ultimately result in execution of machine language instructions on a hardware processor. The machine language is all the processor "understands." However, this language is difficult for humans to comprehend, so most programs are developed in a higher level language that provides a layer of abstraction. Typically, higher level languages are more portable than lower level ones, since they go through translations before becoming machine instructions. While a machine language only runs on one machine's processor, a C program can run on many different systems with the only effort being recompilation on the host machine. The higher level languages often utilize the features of the operating system. Different

operating systems usually provide similar features, but in different ways. For example, UNIX and Microsoft both support sockets, but each uses a different implementation. Utilizing these features in a high-level language like C requires different function calls for each operating system. These calls have to be modified to port the software to each system. Often other programs are created to facilitate these changes. An example is the Autoconf program that is used to discover differences between environments and automate the changes required to port C source code [Friesenhahn97].

In addition to these impediments to portability, the requirements of the program need to be considered. Clearly, some programs do not have a wide application, such as one that controls a missile's guidance, while other programs have an almost universal use, such as text editors. Therefore, a program should have a need to run on the receiving environment before making the effort to port it.

2.2 Desktop Manager Programs

Graphical User Interfaces and window manager programs are found on most desktop computers today. Macintosh is credited with creating the first computer offering a GUI that was widely used in homes [Hargh86, page 2]. The operating system provided the user with a graphical method of interface rather than a command line that was typical for small

machines at that time. This introduced a new paradigm of user interaction with the computer. A new terminology, WIMP (windows, icons, menus, and pointers) describes many of these concepts [Rubin88, pp. 94-106]. Definitions of WIMP terms are provided in Appendix A. A key area of WIMP is the window concept where a program on the monitor is identified by a rectangular border. These windows are hierarchical in nature allowing an application's parent window to display one or more child windows.

The windows interface allows multiple programs to be visible on the computer's monitor simultaneously, each one usually in a separate window. Desktop or window manager programs were created to manage the program resources that are displayed on the computer screen. They allow the user to open, close, resize, and relocate programs.

The desktop paradigm is that the computer monitor represents the top of the user's desk and the computer programs represent papers on the desktop [Evans96]. Using the window concept, the programs, like papers on a desk, can be repositioned and can totally or partially cover, or be covered by other programs.

The virtual desktop paradigm is that a bank of desks surrounds a user only allowing access to one at a time. The computer monitor shows the contents of the current desktop

[Evans96]. A Virtual Window, a grid composed of multiple logical screens, is displayed on the monitor. The user selects which logical screen, each representing a desktop, to display on the monitor [Husain96, page 459]. These virtual features aid in program management, by allowing many programs to be easily accessed without cluttering the screen. Figure 1 is an example of an X Windows manager with a virtual window shown in the upper left corner.

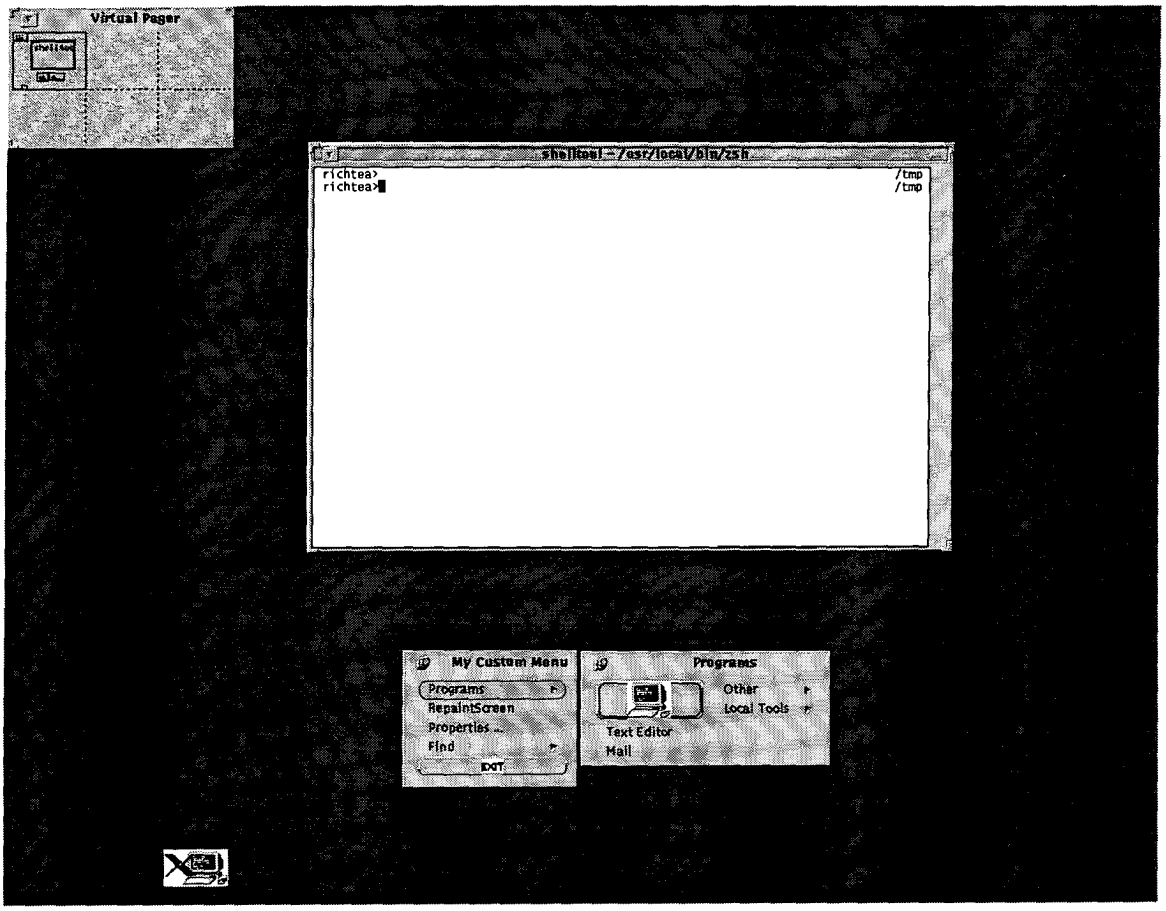

Figure 1: X Windows Desktop Manager

GUI programs also support event driven programming. The user generates mouse and keystroke events that control program

operation. This user control contrasts with procedural programs that have one starting point and then continue to completion.

2.3 Survey of Existing Programs

Macintosh, Microsoft Windows, and UNIX operating systems are three operating systems found on desktop computers. These all support the GUI and desktop paradigm. As previously stated, Macintosh widely used in the educational community, was the first windowing interface provided for desktop computers.

Microsoft, the dominant operating system of business desktop computers, originally used a command line operating system (Disk Operating System or DOS). As processors grew more powerful and less expensive, Microsoft created a windows system GUI that ran on top of DOS. This was later replaced by their Windows 95 and NT systems which were designed as windowing systems and contain an integrated desktop manager.

Both Macintosh and Microsoft Windows are proprietary operating systems containing desktop managers integrated into their operating system. Both these systems were designed with GUI interfaces and the integrated desktop manager provided cannot be disabled. Microsoft Windows allows many of the operating system commands to be accessed

either with a GUI utility program started from the desktop or from a command line interface. This is not surprising since the GUI evolved to replace the command line interface on that software vendor's system. Macintosh does not offer a command line interface. Neither Macintosh nor Microsoft offer virtual features with their window managers, although several add-on virtual managers are available.

The X Window System was created at MIT in the 1980s to provide a network-transparent and vendor-independent GUI for workstations. It is architecturally neutral and allows dissimilar machines to communicate. This system consists of the X protocol (communication), the Xlib (library of subroutines), and X toolkits (for application design) [Jones89, pp. 2-5]. X Windows is multiple software programs that communicate using a standard protocol. X Windows uses the client server model where the X server controls the display and X clients are applications that are served and presented on the display. Both transmission of events and graphic information are supported. X Windows also allows one server to control multiple monitors.

By conforming to the X Windows standard protocol, X Windows applications can communicate with the X server that can run on a different machine. These programs are typically written in C and must be ported to the machine where they run.

Porting these programs requires recompiling on the host machine and possibly some code changes.

X Windows is typically the GUI used on UNIX systems. A variety of X Windows window managers (FWM, MWM, OLWM, TWM) are available for UNIX systems. Many of these are free while others are sold by companies who also produce a version of UNIX. Additionally, most offer virtual features. Although X Windows can be ported to many platforms, it is not commonly used on the Macintosh or Microsoft Windows operating system.

A wide selection of window manager programs exist that all have many similarities [Hopgood88, page 72]. Despite these similarities each has a different look and operation and is closely coupled to the underlying operating system. Since none of them are commonly available on all the platforms mentioned, the default window manager of the operating system is usually used. Consequently each platform presents a different interface to the user.

2.4 Standardization

Standards are one method used to facilitate portability. Two general approaches have been taken: one is the development of open standards, the other the development of proprietary standards [Tanner96, page 88]. The X window system is an example of the former, while Microsoft's Win32 API is an

example of the latter. The API (Application Programming Interface) is also a type of standard that facilitates portability when multiple vendors agree to use the same API [Rowley96, page 80].

Standards originated for hardware and were later applied to software, as for example, the ANSI standard for the C language. There are many non-proprietary standards: national, international, governmental, and industrial. Examples include ISO, NIST, ANSI, and IEEE. Also many manufacturers have developed proprietary standards.

Standards can improve portability by removing differences between systems. They can also create problems if they are ambiguous or manufacturers do not adhere to them. Some standards have succeeded in increasing portability while others have not [Rowley, page 82]. Many people continue to support standards as is evident by the new ones that are continually created.

Commercial vendors and independent organizations have both demonstrated the desire for standardization. Microsoft made efforts to standardize their desktops to have Windows 95 and Windows NT present the same GUI to the user [King94, page 96]. Sun Microsystems made efforts to standardize the desktop user interface with their Common Desktop Environment [Evans96]. The Common Open Software Environment specifies

Motif as the standard interface for UNIX [Husain96, page 397]. These attempts to standardize desktops that are not available on all operating systems have a limited scope. However, the development of a Java desktop manager that could run on many operating systems without code modification would give the user a consistent presentation regardless of the operating platform.

2.5 Java Programming Language

Java was originally developed for consumer electronic devices that use many different microprocessors. This resulted in an architecturally neutral language that made it ideal for use on the Internet, which is a heterogeneous collection of computers that communicate using a common protocol [Flanagan97, page 4]. The Java Applet (a program that executes within a host browser program) demonstrated the language's cross platform capabilities on the Internet. The same Applet program, often containing animation, could execute in a web browser running on many different platforms [Gosling95]. The capabilities of the language have continually improved since its creation. Now, many programs, not just Applets, are written that execute on multiple platforms without modification.

Java achieves architectural neutrality by executing on a Java Virtual Machine (JVM). Java source code is compiled

into Java byte code that is interpreted by a JVM. Java's motto "Write Once, Run Anywhere" really means run anywhere there is a JVM. Java programs only run on one machine, the JVM. This virtual machine isolates the byte code from the underlying environment making compiled Java programs portable to wherever a JVM exists. Figure 2 illustrates the architecture of the JVM.

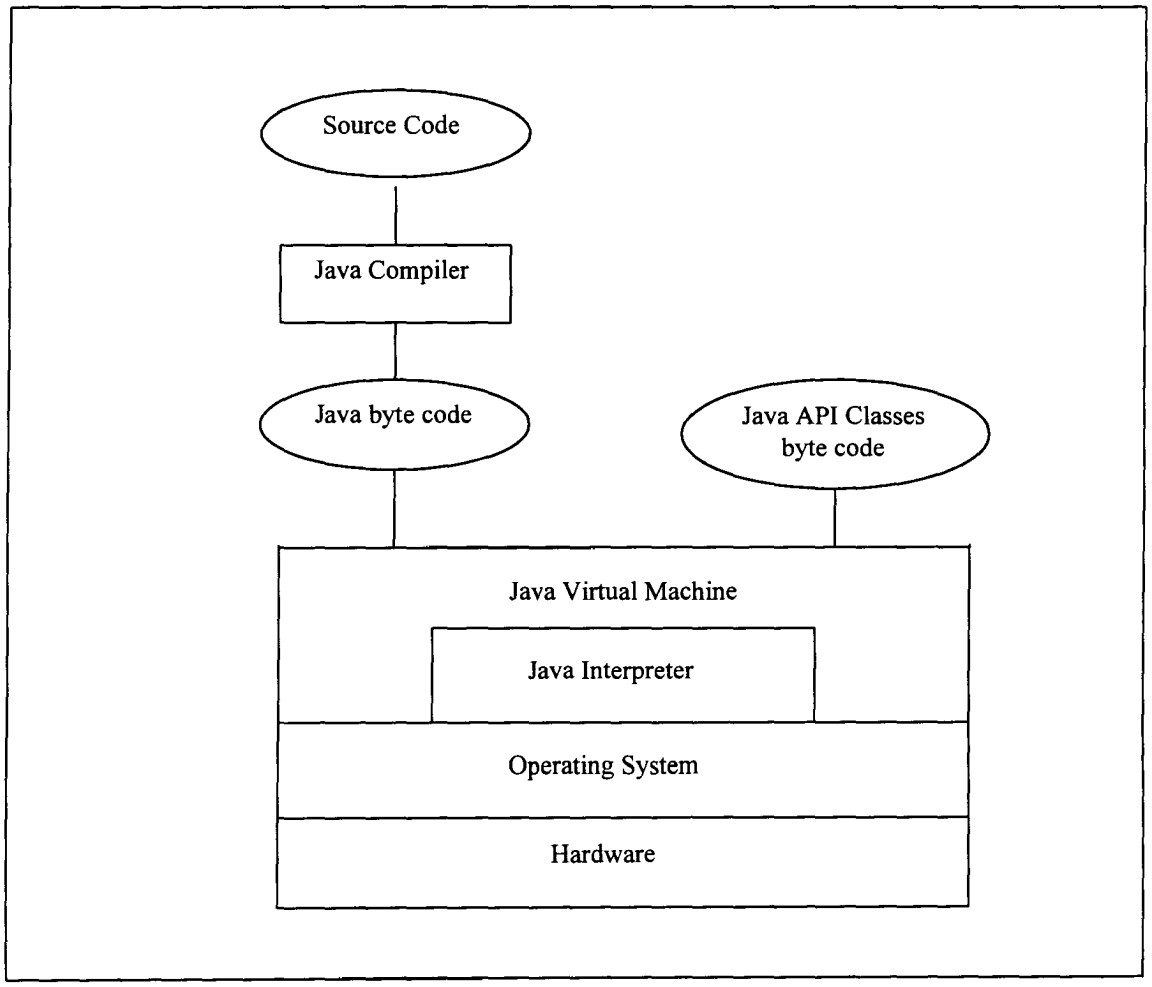

Figure 2: Java Virtual Machine Architecture

The Java Language is a high-level language providing abstraction from the underlying operating system and hardware. The language is 100% portable since it only runs on one machine, the JVM. No changes or compilation are required to run a program on any machine where the JVM exists. This is the appeal of Java, a powerful language that can be ported without changes. Portability comes at the expense of other programmers providing the JVM implementation and class package, certainly not a trivial task. Java programmers reap these benefits since this work has already been accomplished.

The Java language is used to create both Applets and application programs. Applet programs are typically loaded from a remote location and execute on a JVM implemented in a browser program. Their access to files and other resources on the host machine is restricted for security. Java application programs execute on the JVM running as a separate process (not connected to a browser) and are not subject to these security restrictions. The distinction between these two types of programs can be subtle and a program can be created that executes as an Applet or an application. All the programs developed for this thesis are Java programs that operate on a JVM outside of a browser.

The Java language is object-oriented and provides many classes that define the Application Programming Interface

(API) of the language. These classes are required along with the JVM to interpret the byte code. The classes are bundled into packages and include AWT classes for windowing and GUI features, I/O classes for file access, and Network classes for sockets. A complete listing of the API classes is provided in Appendix B. These classes provide the power of the language, as they encapsulate the implementation details. This allows the programmer to concentrate on the program's desired function.

Java's architectural neutrality only applies to the programming language, the JVM is platform dependent, but has been ported to Macintosh, Microsoft's Windows, and UNIX operating systems thus allowing the same Java compiled code to execute, without changes, on different machines [Gosling95, section 1.2.3]. Additionally, the JVM has been implemented as software that runs at the machine level in hardware used in portable electronic devices. This thesis will only be concerned with the JVM implemented on top of an operating system. The Java platform hides many of the issues of portability stated in the previous chapter. Moving the portability issue to a layer below the Java language allows a programmer to develop programs that can run on many different platforms without additional effort.

The implementation details of the Java Virtual Machine are beyond the scope of this thesis, but are specified in a

document by Tim Lindholm [Lindholm96]. Sun offers implementations of a JVM that run on top of the three operating systems previously mentioned. Others have developed implementations for additional systems like Linux. This thesis will not be concerned with these portability issues, because from the programmer's perspective, the JVM is the same consistent platform regardless of where it resides.

Java's device support is limited to satisfying the functionality of the language. Presently devices such as mice, graphic monitors, networks, sound cards, and printers are supported. As Sun introduces new features, like speech recognition and fax processing, the necessary class and virtual machine changes are also provided to support these enhancements. Accessing an unsupported peripheral device can be accomplished using native methods and C program calls. Doing this requires porting the C code along with the Java program that uses the device.

Java imposes requirements on the operating system such as support for multi-threading. Java programs using GUI features require the operating system to support GUI displays. Both Macintosh and Microsoft systems support GUI, but UNIX requires additional X Windows programs. Therefore, Java GUI programs executing on UNIX must start in an X Window to function correctly. Since implementations of X

Windows are commonly found wherever the hardware supports a GUI, this will not be considered a restriction. Other issues such as data representation and file operations are hidden by the JVM, so the programmer does not have to consider these issues.

The Java language allows access to the operating system and supports programming native methods that allow access to executable programs written in other languages such as C. Native methods limit the portability of the Java program because the local code also has to be ported along with the Java code. While the Java code can be ported without any changes, C programs usually have to be compiled on the target machine and often require other changes.

The Java API classes, included with the implementation of the JVM, provide the functionality and power of the language as explained in the next chapter. Often programmers create new class packages providing features not supported by the API. For example ObjectSpace, Inc. created classes to support sorting and searching using different algorithms. The programmer assumes that the Java API classes will be available wherever the JVM runs, but must provide all other classes necessary for program execution. Java provides tools for packaging and transporting new class packages with a program.

Currently no approved standards exist for the Java platform. Sun Microsystems provides most implementations of Java, but has licensed the technology to other companies. Sun tries to enforce standardization, but this is not always possible as evidenced by a suit Sun brought against Microsoft in October of 1997 [Sun97]. Sun Microsystems is attempting to make the Java Platform a standard by registering it with the ISO/IEC JTC 1 (International Organization for Standardization / International Electrotechnical Commission Joint Technical Committee 1) as a Publicly Available Specification (PAS) [Sun97A]. In addition Sun has established a procedure for programs to be certified as "100% Pure Java" assuring that they will run anywhere the JVM exists [Sun98]. This is a voluntary procedure, but developers whose software passes these tests label it as "100% Pure Java".

Chapter 3

PROGRAM REQUIREMENTS

The requirements of the Java desktop manager program will be specified based upon the features of the manager programs that exist for the three operating systems previously mentioned. The utility programs were selected to provide platform independent features that are normally included with a desktop manager. Some of these programs allow customization of the manager, while others provide access to system resources. Since these Java programs are portable, they do not provide platform specific features.

3.1 Desktop Manager

A desktop manager has two functions: one is to manage the presentation, the other is to manage the operations. The presentation is concerned with window size, layout, and maintaining an iconic representation of the window. The operations respond to user input from mouse and keyboard actions [Hopgood88, page 67].

The requirements for the Java desktop manager with virtual features are as follows:

- Be portable to a JVM running on a Macintosh, Microsoft, or UNIX operating system.

- Present a uniform background to the user with a virtual window on the monitor, composed of multiple logical screens, that allows the user to select the logical screen to be displayed.

- Allow the user to change virtual window features of size, location, number of logical screens, and color.

- Provide a pop up menu to allow the user to select programs to open. Programs shall be able to be loaded both from local disk and remote computers. Local programs shall be selected from a file that contains the available application classes.

- Place an icon representing each open application on the logical screen representing the current view. The program's name will display when the cursor is positioned over its icon.

- Respond to the application's window movements and size changes by changing its icon location and size. Display the icon of the application in focus differently from all other icons.

Java provides many features and classes that support the program design. The first design decision is selecting the method of communication between the desktop manager and the applications since this is critical for program operation. Java provides the following three mechanisms [Campione97]:

1. Object methods - One object calls the method of another object. Java is object oriented and supports the event driven programming model using event classes.

2. Sockets - Java includes network classes that allow access to sockets through the operating system.

3. Applets - Multiple Applets running on the same browser are able to communicate through the JVM.

The object model is the best choice for communication in the desktop manager, since it provides the most flexibility. This will allow asynchronous communication, in response to user key and mouse action events. Additionally, this model allows all the applications to be Java programs rather than Applets.

The socket model was not selected, because it adds additional overhead to the communication mechanism. Sockets would be better suited if a distributed desktop manager were

being developed, but this manager will only control programs running on one computer.

Requiring programs to be Applets forces them to adhere to the Applet security restrictions that limit file and socket access. Additionally, it would require the desktop manager program to be a browser, placing further restrictions on the program. Therefore, this model was not chosen.

The application and utility programs to be managed will be Java Programs using GUI features. Java provides an Abstract Windowing Toolkit that is a set of classes which support the windowing requirements and the WIMP (Appendix A) model. These classes are used to control the desktop display and provide iconic representations of the open programs.

The virtual window is shown as sticky or present, regardless of which logical screen is displayed, so that a mouse click on it at anytime provides access to the other logical screens. If this were not the case, an application window could cover it and prevent the user from accessing it [Husain96, page 461]. Java has classes that support these requirements.

Java programs run on the JVM, which runs as one process on the operating system. Java programs run in a window on systems that have an integral window manager. The desktop

manager will try to hide this by maximizing its window to cover the screen and hide the existing manager. Additionally, this should prevent mouse events that change window focus from reaching other windows, thereby allowing the desktop manager to handle all these events. It would be preferable to remove the existing manager, but since this program is being developed to investigate Java and portability issues, this inconvenience is accepted.

3.2 Utility Programs

Utility or accessory programs such as a clock, calculator, file manager, games, terminal window, and text editor are usually available on desktop computers. These programs are included in order to evaluate their portability requirements.

The clock requires knowing the date for which Java provides a class that obtains this information from the operating system. The clock is displayed as both a face with hands, and in digital format.

The calculator uses GUI features to allow the user to perform standard and scientific (exponential, logarithmic, and trigonometric) calculations. Java's AWT and Math classes support these functions.

A file manager program provides access to the file system displayed as a tree structure. The Java IO class supports local file access. The Java RMI (Remote Method Invocation) classes provide methods to access remote files by allowing local objects to invoke methods of remote objects using sockets for communication. A security manager prevents unauthorized access to remote files. Recently released Swing AWT classes support a graphical display of the files.

A terminal window or emulator is a feature provided by a GUI manager whose underlying operating system supports command line execution. Both UNIX and Microsoft Windows provide this, but Macintosh does not. While this is not the same as executing native methods, it allows the program user to execute OS specific commands. Obviously not all operating systems support the same commands, therefore unsupported commands will generate errors. The program will present those errors to the user, but continue to operate. The Java System class provides the capability to deliver this feature.

A text editor utility is provided as an example program with Java's Swing release. This program features selectable fonts and printing capability.

The Tic Tac Toe game is provided as a program example with the Java release. It utilizes Java's ability to display images and respond to user events.

Chapter 4

DETAILED PROGRAM DESIGN

The Java language is young and continues to evolve with a 1.1 release in 1997 and a 1.2 release in 1998. Several major changes, such as Version 1.1's introduction of a new event handling model and Version 1.2's introduction of a new GUI component model, resulted in changing the design of the desktop manager during development. This chapter details how the Java classes support the desktop manager's program design.

4.1 GUI Features

The Java Abstract Windowing Toolkit (AWT) classes support the GUI features of the desktop manager program. During program design the Java AWT underwent major changes when the "Swing" classes were introduced as part of the Java Foundation Classes (JFC) release. The desktop manager program uses many features offered by these newer classes.

AWT classes were originally implemented by accessing the native AWT classes (Java calls them peer classes) of the host operating system. This resulted in a Java widget, the term used for GUI entities like buttons, labels, and text

boxes, being implemented using the widget of the operating system underlying the JVM. Although the Java code would run on all JVMs, the GUI component's presentation would be determined by the operating system, not the program. This did not compromise the "run anywhere" motto, but it did show Java's dependence on the operating system.

Swing provides a different AWT component implementation that is considered "light weight," because it is not tied to the underlying operating system through peer classes. This is accomplished by using a model that supports what Java calls a "pluggable Look and Feel" (L&F) [Sun97B]. The Look refers to how a widget is presented on the screen. A button, for example, has a different border on Microsoft Windows than on UNIX. The Feel refers to how the component responds to user actions. For example, when "pressed," the border of a Microsoft Windows button changes, while both the border and color of a Motif button change.

The concept of Look and Feel is not usually addressed as a portability issue. Java programs are considered 100% portable even if they do not "look" the same on all JVMs. Other software, like X Windows, has the capability of displaying a platform independent look, as demonstrated by the FVWM95 desktop manager that provides a Microsoft Windows 95 look while running on UNIX platforms. Since the Swing classes were introduced, the appearance and actions of Java

programs no longer have to be tied to the native operating system where they run. Swing GUI components can Look and Feel the same on all JVMs, subject of course to copyright restrictions [Sun97B]. This feature enhances Java's platform independence by shielding the user from the underlying environment. A Motif look, as shown in Figure 3, can now exist on a Macintosh, Microsoft Windows, and UNIX operating systems. Programmers can also develop a custom look not associated with any specific platform. An example of Java's custom look is shown in Figure 4.

Java's pluggable Look and Feel is accomplished by using a variation of the Model/View/Controller (MVC) architecture

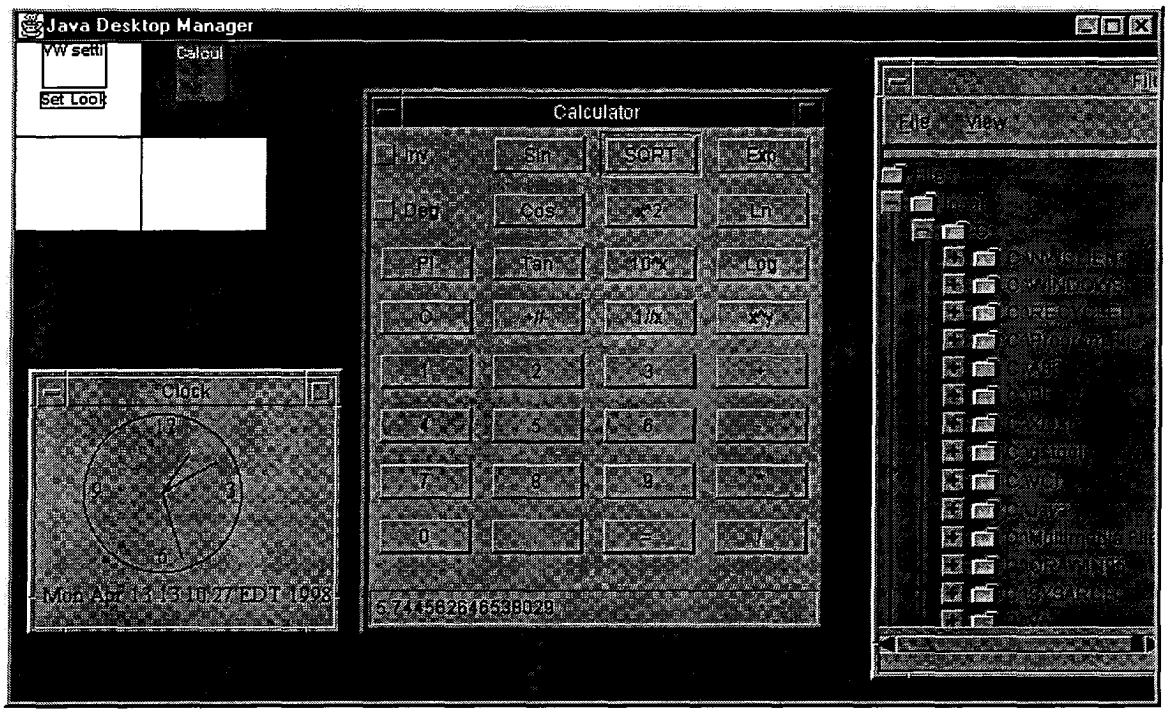

Figure 3: Desktop Manager with Motif Look

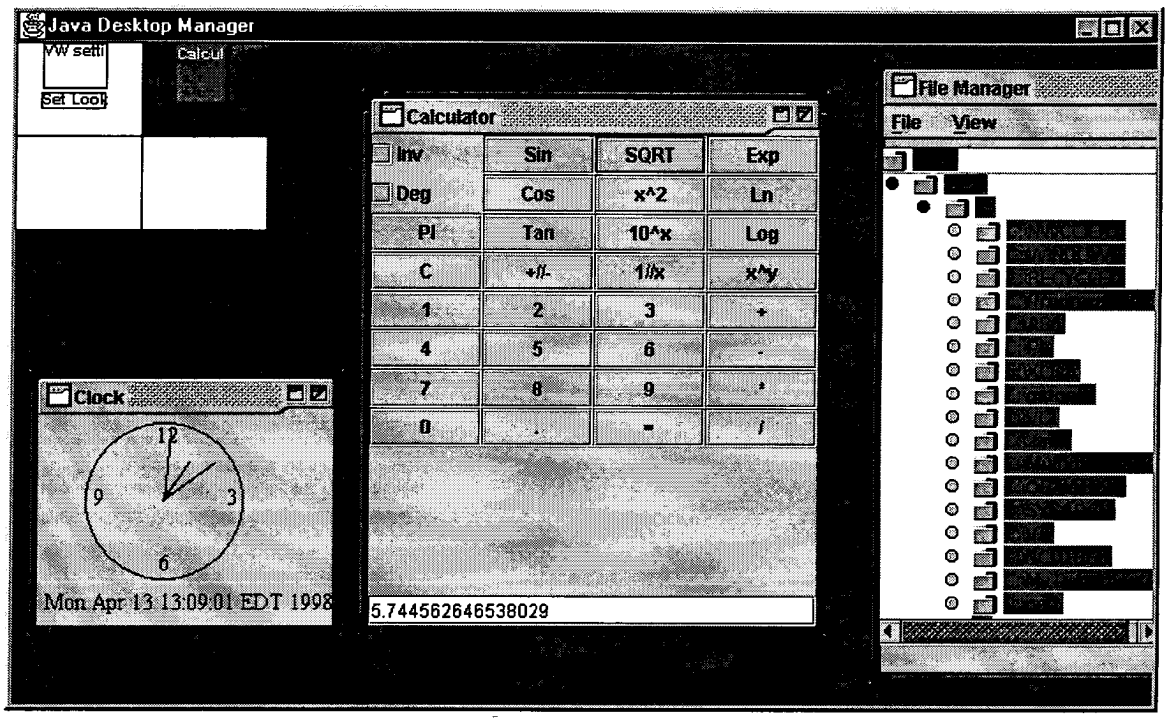

Figure 4: Desktop Manager with Java Look

used by Smalltalk which was developed at the Xerox Palo Alto Research Center [Yourdon94, page 272]. The three elements of the architecture control all aspects of an object. The Model part determines the object's behavior, the View (the Look) controls the presentation, and the Controller (the Feel) changes parameters in the Model in response to events. The View and Controller are platform independent, allowing widgets to be developed without relying on the operating system.

The MVC architecture provides two other benefits. The first is the ability to change the View and/or Controller while the program is running. The other is the ability to create multiple views of the same model, allowing a single object

to be presented in different formats [Sun97B]. The File Manager Utility uses this feature to display the list of file elements as a tree, if they are directories, or as a table, if they are files.

Sun modified the MVC architecture by combining the View and Controller into a single User Interface (UI) object called the delegate while retaining the Model as a separate object. Since the Look and Feel of a component are closely related, changing one often effects the other and requires them to communicate. Combining the View and Controller simplifies this communication allowing changes to either to be made more efficiently [Sun97B].

A Swing GUI widget consists of three parts as shown in Figure 5.

1. The widget object that sets the model and delegate objects.
2. The model object that determines how the widget operates by maintaining and changing its attributes.
3. The delegate object that has the methods that control the widget's presentation and reaction to events.

Java uses the term frame for what has been referred to as a window in this thesis. Java applications use a frame class to display themselves. A frame is implemented as a peer component and is subject to several restrictions. It cannot

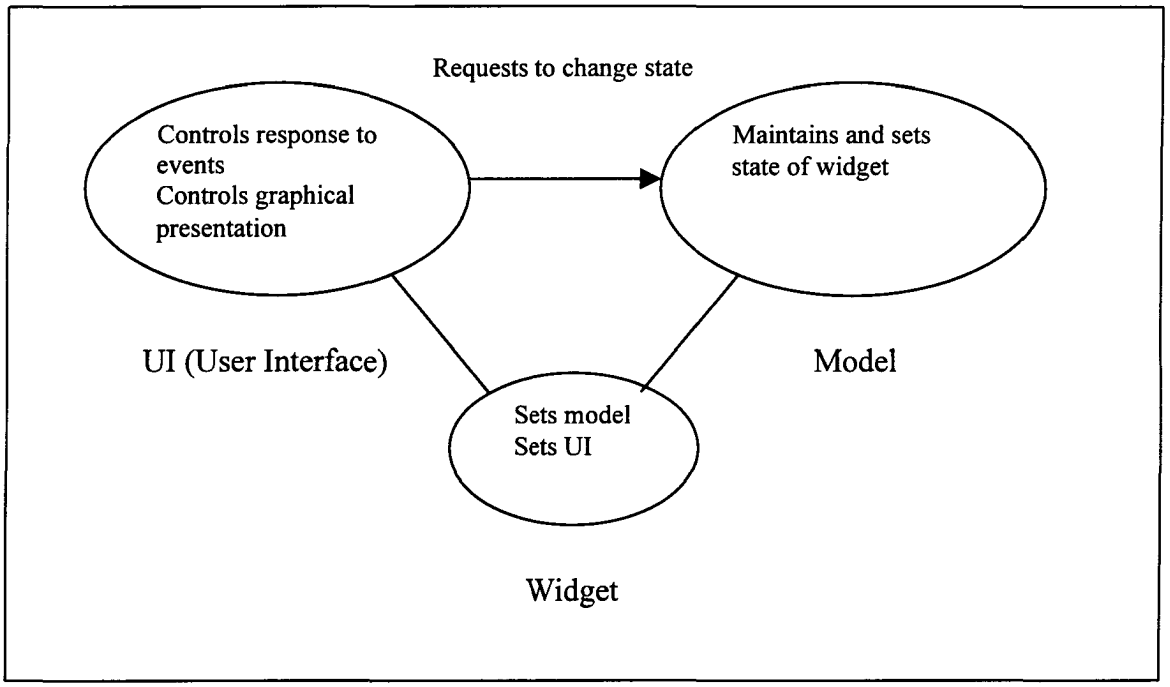

Figure 5: Model View Controller (MVC) Architecture

contain other frames, which hinders the main objective of the desktop manager - displaying and controlling multiple applications that are enclosed in frames. Swing introduced an "internal frame" class that is independent of the host operating system and is not constrained by the restrictions of frames. The desktop manager uses this feature to contain and control multiple utility programs that are subclasses of internal frames.

Internal frames also facilitated the creation of a "sticky" virtual window. Since the virtual window must always be visible for the user to select the logical screen to view, another window should not cover it. However, in an

overlapping window environment this could happen, if another window in focus were dragged over the virtual window. The Swing classes allow setting the viewing hierarchy of windows, thus enabling the program to control which window is in the foreground covering other windows. The virtual window is set so no other window can cover it.

Java animation often flickers because the mechanism used to "paint" the screen first clears the screen before drawing an image. The Swing components incorporate several of the techniques developed to reduce flicker. Using those components improved the presentation of the desktop manager.

4.2 Event Model

Java is an object-oriented language that provides support for event driven communication between objects using event classes. The desktop manager employs this feature to control the utility programs and the virtual window.

Java's event model uses the terminology of sources and listeners. Objects that generate events are sources and objects that take actions in response to events are listeners. An object interested in a specific type of event notifies the source object by registering itself as a listener. The source object keeps a list of these listener objects and when an event occurs, it calls the method

associated with that event for every object in the list. This is typically done asynchronously without any priority given to the listeners. Swing classes provide methods to access the event queue for synchronizing events.

The programmer can access events at two levels. This will be illustrated by using a mouse click on a button. At the low level, the click is a mouse event at a specific location on the screen, but at the high level it is a button press. The programmer accesses the event at the low level by overriding the mouse action method of the button's parent AWT component object. The high level event is accessed by overriding the button pressed method of the button object. Objects can listen for low or high level events or both by registering with the desired event's source object.

Utility program actions that move or resize their windows require the desktop manager to take actions to maintain program icons on the virtual window. Additionally, user selection of the logical screen to display requires displaying the utilities on that screen. Prior to the Swing class release, this was accomplished by the utility programs generating a custom event for each of these actions. The desktop manager listened for these custom events and then responded with the required actions.

Swing classes introduced a manager class, based on the MVC concept, that receives all window change events from its children. The desktop manager uses this class and overrides its methods to take the necessary actions. This eliminated the need to modify the utility programs to generate custom events for the desktop manager. The desktop manager can control any Java utility program that is a subclass of the internal frame class as shown in Figure 6.

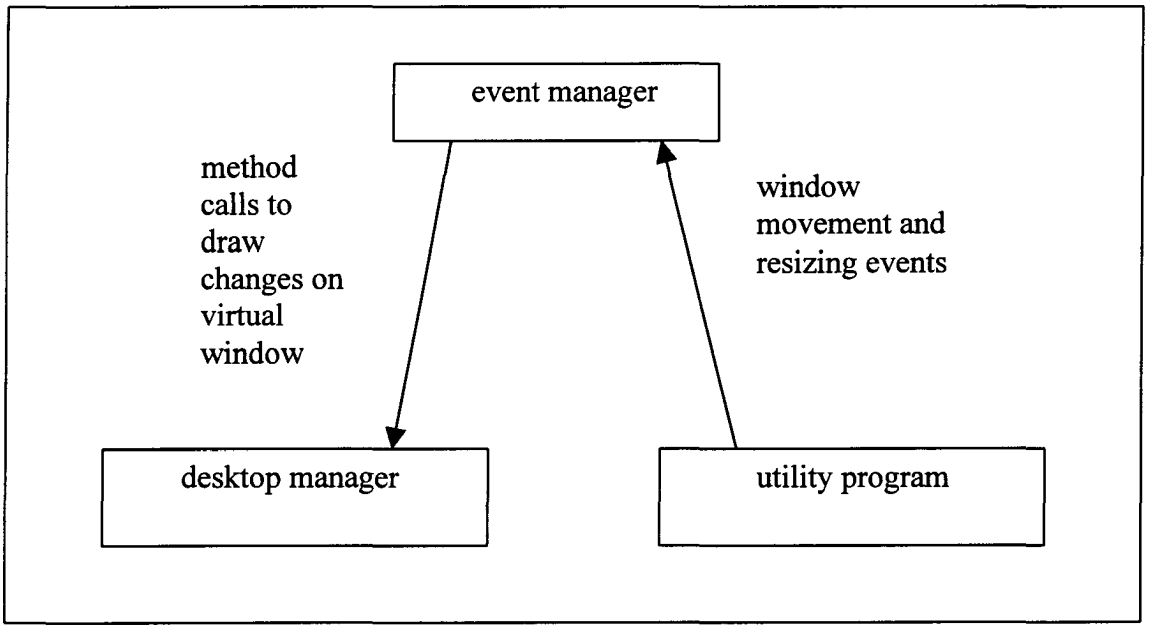

Figure 6: Desktop Manager Event Diagram

4.3 Class Loading

Java programs consist of one or more classes (compiled to byte code) that are loaded into a JVM and instantiated to objects. The JVM uses a "class loader" to load all classes dynamically as they are required. Starting a Java program

(not an Applet) requires starting a JVM that then loads a "public" class and invokes the "main" method to start a program running. The JVM does not run by itself without a Java class, but requires this "public" class argument when it is started. Additional programs or classes are loaded as necessary and all run on the same virtual machine as different threads.

The desktop manager is the first class loaded after the JVM is created. Subsequently the utility programs are loaded as the user selects them. When the JVM is requested to load a class, it searches to see if the class has already been loaded. If it has not, a default class loader is used. The programmer can override this feature and define a loader to load files from local disk or across networks using sockets [McManis97]. The desktop manager implements this feature to allow access to both local and remote programs.

This class loader provides a consistent model for starting all programs and provides security by only allowing classes to be loaded from specified directories [Sun97C, section 9]. This is necessary since connecting a machine to a network introduces the possibility of external programs corrupting the system.

4.4 Remote Object Access

The class loader and file manager programs require access to resources on remote computers. Java introduced the Remote Method Invocation (RMI) feature in the 1.1 release that allows access to the methods of remote objects. RMI provides transparent access to an object's "public" methods across a network, hiding the details from the programmer.

Other vendors provide similar ways to access remote resources either as remote procedure calls or remote object requests. COBRA, which Java supports, is a specification allowing objects from different systems to communicate [Morgan97]. RMI is used for the desktop manager, because it uses only Java and is 100% portable to any JVM.

Remote access typically requires communication, translation of data types, marshaling of arguments, and security. RMI handles all these using Java code. Sockets are used for communication while translation of data types is not required, since both the local and remote objects are on the same type of machine, a JVM. Since Java is object oriented, the data transferred is an object. The contents of an object are marshaled using the Java serialization technique which compresses all the attributes and methods of an object into a stream [Flanagan97, page 172]. The receiver decompresses the stream to recreate the object. Security is provided by

only allowing access to objects from locations approved by the RMI security manager object. Additionally, serialized objects contain the URL of their origin, so that if additional objects are required, they are only permitted from that location [Sun97C, section 9].

RMI operation requires the remote objects to register with a RMI server that listens for all requests. When a request is received, the server finds the registered object and establishes the communication between the client object and the remote object. A stub object on the client side and a skeleton one on the server side handle the communication and transfer of information as shown in Figure 7.

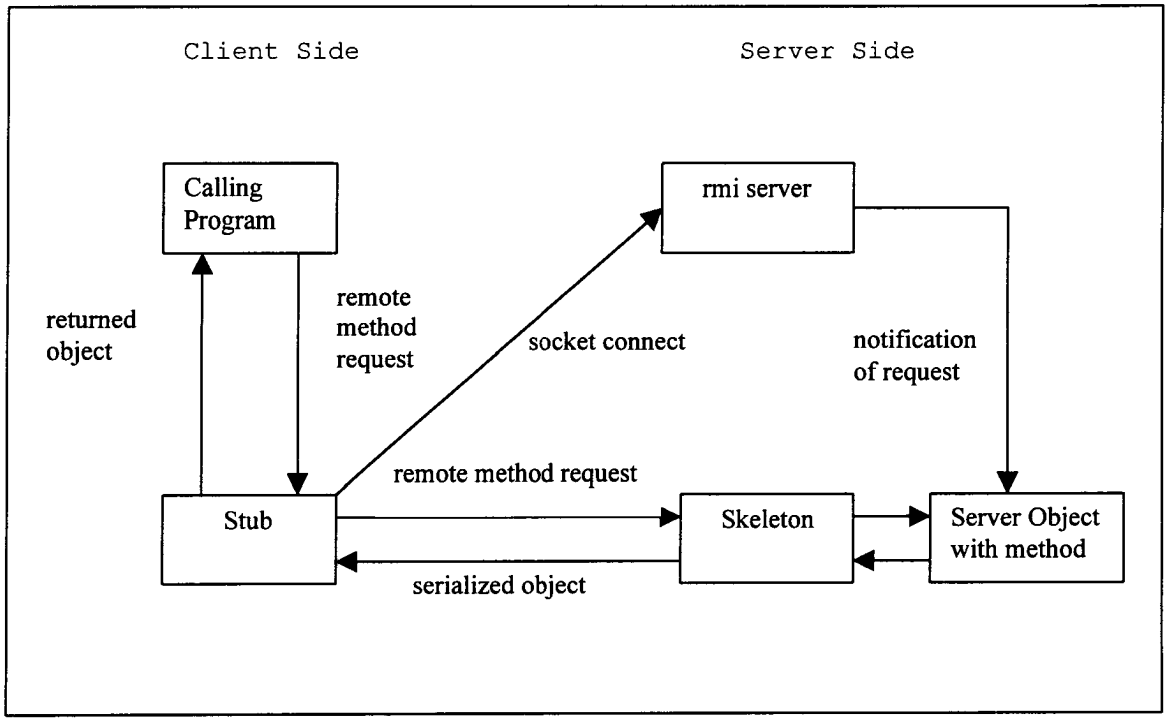

Figure 7: Remote Method Invocation (RMI) Operation

4.5 Operating System Access

Java programs execute on the JVM that normally shields them from the operating system. However, the JVM does provide access to the OS at several different levels as shown in Figure 8.

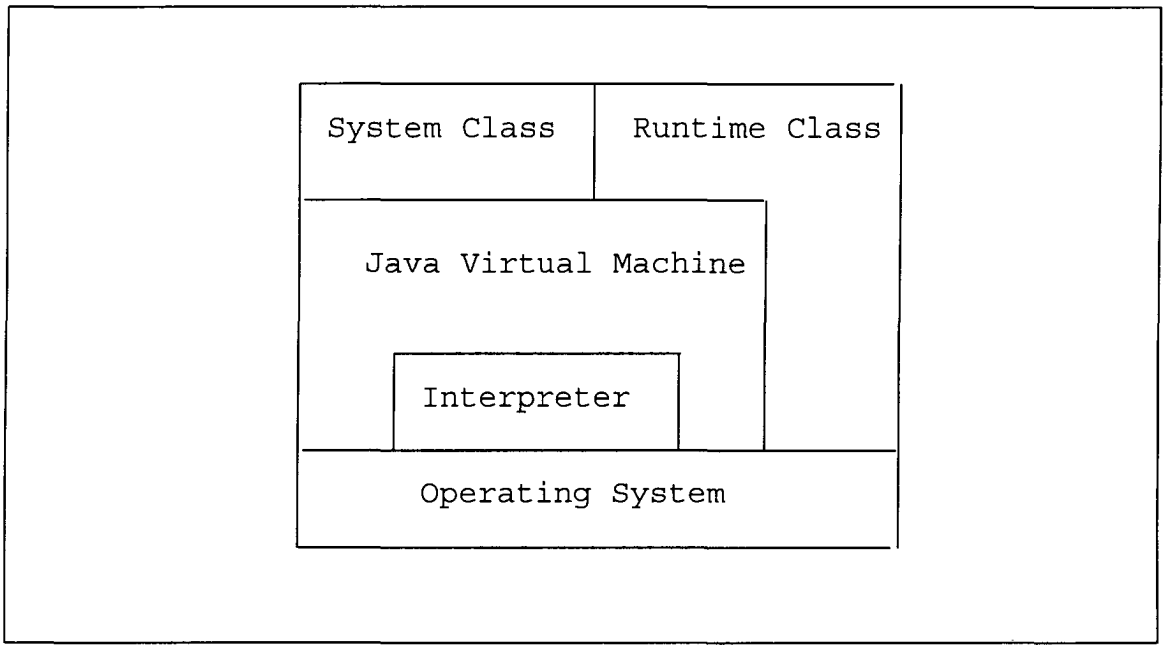

Figure 8: JVM Operating System Access

At the highest level of abstraction, the System Class allows access to parameters like the user's home directory, type and version of the OS, Java class path and directory, date, and time. The same methods are used to obtain this information on all implementations of the JVM. From the programmer's perspective these are 100% portable because

they are available wherever the JVM exists. The clock utility program uses the date and time.

Java's IO class provides similar parameters for file and directory access. For example, the path separator character that is different on UNIX and Microsoft Windows, is provided as a system parameter. Additionally, creating, listing or modifying files or directories are all performed with Java methods that are platform independent. Access to the disk structure is, however, platform dependent. Microsoft Windows uses different terminology than UNIX for physical and logical disk drives. The file manager utility used a switch statement to execute different code, depending on the underlying operating system. This makes the code only portable to the three operating systems mentioned previously.

At a lower level, the Runtime Class provides direct access to the operating system, allowing Java programs to execute system dependent commands. The commands to be executed are first written to a file, then executed by creating a process separate from the JVM using the "exec" procedure. The Runtime object maintains contact with the new process by accessing stdin, stdout, and stderr. The Terminal Window utility was developed, using the Runtime class to provide the user with the ability to access the underlying operating system with text commands.

Java supports running of other language programs, like C and C++, on the operating system. Called native methods, these are often used to improve program performance or provide a feature not available in the Java language. Neither the desktop manager nor the utility programs use any native methods, because they negate the objective of having an 100% portable program.

Chapter 5

CONCLUSION

The desktop manager program was developed and tested on a Windows NT operating system. The compiled byte code from the development machine was executed on a JVM running on a Windows 95 (figure 9) and on a UNIX platform demonstrating this Java GUI program to be 100% portable. A look at each of the four areas of portability will provide details of how the restrictions were overcome.

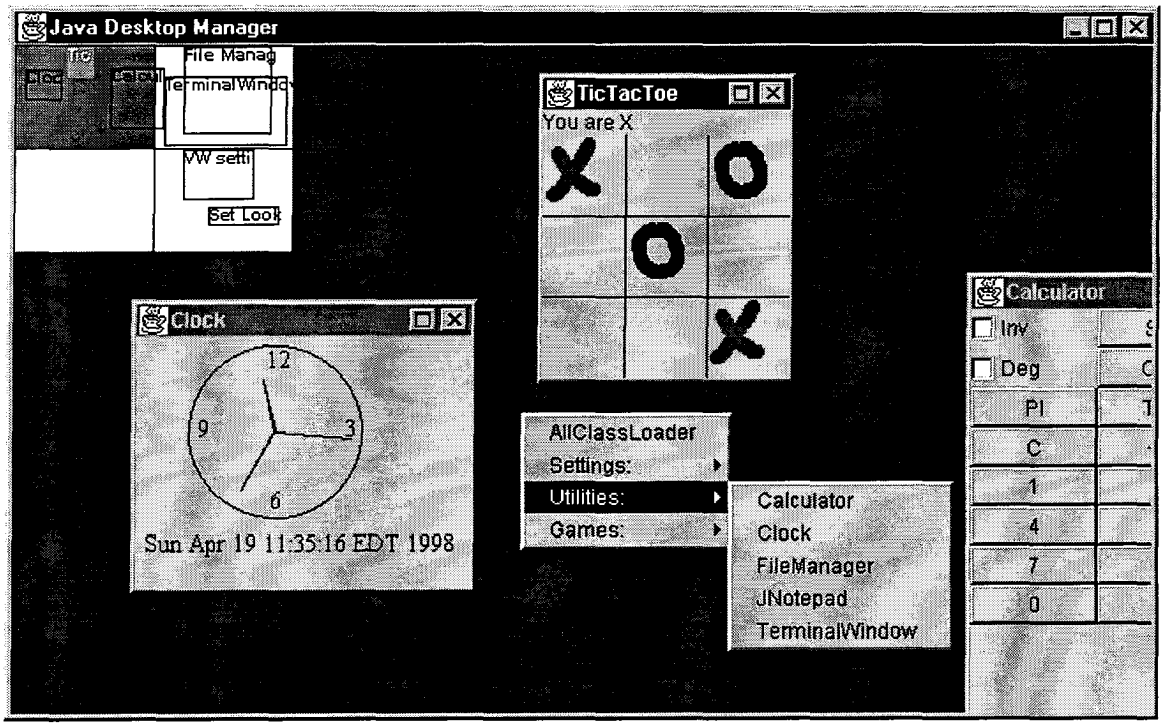

Figure 9: Desktop Manager with Windows Look

Determining the program requirements before choosing the language and platform identified portability restrictions. This design conceded that the desktop manager would only run where Sun has already ported the JVM and only use hardware supported by the existing classes. Considerable effort would be needed to program a new peripheral device, as this would require different native code for each platform where Java runs. Java supports the devices required by the desktop manager and continues to create new classes that provide additional features. Therefore, this constraint did not limit the development or features of the desktop manager and ensured 100% portability.

Java programs execute on the JVM that runs as one process on an operating system. GUI environments provide a window for starting the JVM from a command line prompt. The desktop manager maximized its display window to cover the entire screen; giving the appearance that it was controlling the monitor. This technique resulted in several difficulties with the existing OS window's manager. On Microsoft Windows, the task bar would cover a portion of the desktop, unless it was set to not be "always on top." On UNIX, X Windows was required, adding another software layer. Also, if X Windows had a virtual desktop manager running, its virtual window would display on top of this desktop manager, resulting in two virtual windows. Using a different X Windows manager without virtual features eliminated this problem. Neither

problem prevented the program from operating, but they did demonstrate that the Java program was just another program running on that system. A desktop manager program should operate at a lower level, to have more control of the display.

The terminal window program demonstrated that Java has limited access to the operating system. Direct access to the underlying OS through the JVM was not obtainable as desired. Since the Java language was designed to run on an architecturally neutral machine, this is not surprising. The terminal window utility was a crude implementation, but it did identify an area where Java does not provide support that is available from other platform specific languages.

The File Manager utility had difficulty accessing the drive structure of the host machine. Java does not provide a class method that obtains this information, therefore an algorithm was developed that executes differently, depending on the underlying OS. The tree structure of the Swing classes provided an excellent format for the display. Java API classes do not support sorting of files and directories, but a third party class can provide this feature.

Recent changes have greatly enhanced the power of the language with features like multimedia, international capabilities, drag and drop, enhanced drawing, a pluggable

look and feel, accessibility, and remote program execution. Many of these changes facilitated the development of the desktop manager. While these changes have greatly expanded the language, they also have created portability problems. Obviously the older versions do not support the features of the newer versions, but the newer classes are not always backward compatible. Specifically, some of the class methods were changed to make them Java Bean (Java's model for reusable software) compliant and older version code will not compile with the newer class libraries. The byte code will execute on the newer JVM, but if changes are needed to the original code, it must be modified before recompiling. Java provides a tool to facilitate the conversion of the older source code, but 100% portability is compromised.

The language contains known bugs. For example, accessing files in the root path of a drive requires different syntax for Windows 95 and Windows NT. Most bugs appear to be a result of a new language trying to quickly provide many features.

The pluggable look and feel was a powerful feature that allowed a graphical presentation independent of the operating system. As computer networks expand, this feature may become more valuable by allowing the programmer to give the user a consistent interface on multiple environments.

Sun Microsystems, Inc. developed the Java technology and licenses it to many users. They also developed the API classes that are supplied with the JVM. As demand for Java enhancements continue, other vendors will develop API classes (e.g. Microsoft AFC classes) that could lead to incompatibilities between implementations. As stated previously, Sun is attempting to standardize Java to maintain its portability and prevent multiple implementations.

While Sun provides many implementations of the JVM, most browsers use their own manufacturer's implementation. Other vendors are also developing JIT (Just In Time) and HotSpot compilers to improve the JVM performance [Armstrong98]. These new compilers increase processing speed by translating the Java byte code into native machine code, thus bypassing the interpreter. These multiple enhancements introduce the possibility that not all JVMs will comply with the standard and compromise 100% portability. However, the desktop manager used only Sun's JVMs and did not experience any problems.

5.1 Future Research

Networks and distributed computing are becoming more common. The paradigm of a desktop computer being self-contained is changing to align with Sun's motto "the network is the

computer." The network computer is one where all the programs, perhaps even the OS, are external to the machine. The Java language supports networking and portability and can provide a solution for this computing paradigm.

While this desktop manager allowed loading of remote programs, that was not its main intent. A desktop manager will be needed for the network computer that may not contain a disk drive. One challenge will be to provide transparent access to remote programs and resources, perhaps including the ability to obtain the JVM remotely. Another requirement will be the capability of accessing remote drives for saving and restoring files.

Since the network computer's hardware resources will be minimal, programs may execute on a server machine and only provide the client with a graphical representation. These remotely executing programs would take advantage of the server's greater resources and the network computer would just provide a window to the process. Development of a desktop manager program to control these remote resources is another area of future research.

REFERENCES

[Armstrong98]
Armstrong, E., "HotSpot: A new breed of virtual machines," Java World, March, 1998, http://www.javaworld.com/javaworld/jw-03-1998/jw-03-hotspot.html.

[Campione97]
Campione, M. and K. Walrath, The Java Tutorial, 1997, http://java.sun.com/docs/books/tutorial/toc.html.

[Corel97]
Corel Corporation, "Corel Office for Java Wins Its First Award," May, 1997, http://www.corel.com/news/1997/may/GONEcomtech97.htm

[Cox90]
Cox, B., "There Is a Silver Bullet," Byte, October, 1990, pp. 209-218.

[Evans96]
Evans, S., "Common Desktop Environment," Sun Microsystems Inc., 1996, http://www.sun.com/solaris/cde/cde-wpaper1.html.

[Flanagan97]
Flanagan, D., Java in a Nutshell, Second Edition, O'Reilly and Associates, California, 1997.

[Friesenhahn97]
Friesenhahn, R., "Autoconf makes for Portable Software," Byte, November, 1997, http://www.byte.com/art/9711/sec4/art2.html.

[Gosling95]
Gosling, J. and H. McGilton, "The Java Language Environment: A White Paper," Sun Microsystems, Inc., 1995, http://java.sun.com/doc/index.html.

[Hargh86]
Hargh, R. and L. Radford, Macintosh Logo, Wiley and Sons Inc., New York, 1986.

[Hopgood88]
Hopgood, D., "Window Interfaces: A Taxonomy," IEEE Computer Graphics and Applications, 8, 5 (September, 1988), pp. 65-84.

[Husain96]
Husain, K. and T. Parker, et al., Linux Unleashed Second Edition, Sams Publishing, Indiana, 1996.

[Jones89]
Jones, O., Introduction to the X Window System, Prentice-Hall, New Jersey, 1989.

[King94]
King, A., Inside Windows 95, Microsoft Press, Washington, 1994.

[Lecarme89]
Lecarme, O., M. P. Gart, and M. Gart, Software Portability, McGraw-Hill, California, 1989.

[Lee93]
Lee, G., Object-Oriented Application Development, Prentice-Hall Inc., New Jersey, 1993.

[Lindholm96]
Lindholm, T. and F. Yellin, "The Java Virtual Machine Specification," 1996 http://sun.java.com/docs/index.html.

[McManis97]
McManis, C., "The basics of Java class loaders," Java World, October, 1997, http://www.javaworld.com/javaworld/jw-10-1997/jw-10-indepth.html.

[Morgan97]
Morgan, B. "COBRA meets Java," Java World, October, 1997, http://www.javaworld.com/javaworld/jw-10-1997/jw-10-corbajava.html.

[Rowley96]
Rowley, D., "The business of Application Portability," StandardView 4, 2 (June, 1996), pp. 80-87.

[Rubin88]
Rubin, T., User Interface Design for Computer Systems, Ellis Horwood Limited, England, 1988.

[Sun97]
Sun Microsystems, Inc., "Sun Sues Microsoft for Breach of Java Contract," October, 1997, http://java.sun.com/pr/1997/oct/pr971007.html

[Sun97A]
Sun Microsystems, Inc., "International Standards Organization Members Approve Sun's PAS Application," November, 1997, http://java.sun.com/pr/1997/nov/pr971117.html

[Sun97B]
Sun Microsystems, Inc., "Swing Architecture (Pre-beta Material)," Version 0.5, 1997, http://www.java.sun.com/jfc/swinf-0.6/doc/overview2.html

[Sun97C]
Sun Microsystems, Inc., "JDK 1.1.5 Documentation," 1997, http://java.sun.com/java/jdk/1.1.5/docs/index.html.

[Sun98]
Sun Microsystems, Inc., "100% Pure Java Program" http://java.sun.com/100percent/cert.html

[Tanner96]
Tanner, P., "Software Portability: Still an Open Issue?," StandardView 4, 2 (June, 1996), pp. 88-93.

[Yourdon94]
Yourdon, E., Object-Oriented Systems Design - An Integrated Approach, Prentice-Hall Inc., 1994, pp. 269-270.

APPENDIX A

WIMP TERMINOLOGY

WINDOWS, ICONS, MENUS, POINTERS

Windows: An area generally defined by a rectangular border in which a program displays information.

Tiled - windows are tiled if they do not overlap each other when displayed on the monitor.

Overlapped - windows overlap if one window covers a portion of another window.

Sizing - changing the window size.

Scrolling - a method that changes the information visible in a window.

Iconifying - replacing the window with an icon representing the window.

Title Bar - area of the window that contains text (and often an icon) identifying the program that owns the window.

Icons: A pictorial representation of a screen object. Usually a bitmap image of the logo or trademark of the program.

Menus: A method allowing user selection of commands by selection of a textual representation.

Menu Bar - a horizontal array of menu choices usually located below the title bar.

Pull down - a menu that appears vertically as a result of selecting another menu item.

Pop up - a menu that appears as a result of a user action such as a mouse click. Their location is not fixed, but usually occurs at the location of the mouse click.

Pointers: A method for selecting objects on the monitor. A cursor identifies the pointer's location on the monitor.

Mouse or trackball - a device that controls the position of a cursor on the screen and is equipped with one to three buttons.

Entered, Exited - refers to the cursor location relative to a specific window.

Up, Down, Click, Double Click - refer to mouse button actions. These usually result in some program action. [Rubin88, pp.94-106]

Appendix B

Java Class Hierarchy Index

Java(tm) Platform 1.1.5 Core API

Java API Packages

package java.applet
package java.awt
package java.awt.datatransfer
package java.awt.event
package java.awt.image
package java.beans
package java.io
package java.lang
package java.lang.reflect
package java.math
package java.net
package java.rmi
package java.rmi.dgc
package java.rmi.registry
package java.rmi.server
package java.security
package java.security.acl
package java.security.interfaces
package java.sql
package java.text
package java.util
package java.util.zip

APPENDIX C

Contents of the Java(tm) Development Kit - JDK(tm) 1.1.5

Taken from readme file of JDK1.1.5 [Sun97C]

- Java Compiler (javac)
  Compiles programs written in the Java programming language into bytecodes.

- Java Interpreter (java)
  Executes Java bytecodes. In other words, it runs programs written in the Java programming language.

- Java Runtime Interpreter (jre)
  Similar to the Java Interpreter (java), but intended for end users who do not require all the development-related options available with the java tool.

- Java AppletViewer (appletviewer)
  Used for testing and running applets.

- Java Debugger (jdb)
  Helps you find bugs in Java programs.

- Class File Disassembler (javap)
  Disassembles compiled Java files and prints out a representation of the Java bytecodes.

- Java Documentation Generator (javadoc)
  Parses the declarations and documentation comments in a set of Java source files and produces a set of HTML pages describing the public and protected classes, interfaces, constructors, methods, and fields. Also produces a class hierarchy and an index of all members.

- C Header and Stub File Generator (javah)
  For attaching native methods to Java code.

- Java Archive Tool (jar)
  Combines many Java class files and other resources into a single jar file.

- Digital Signing Tool (javakey)
  Manages entities, including their keys, certificates, and the trust associated with them.

- Native-To-ASCII Converter (native2ascii)
  Converts a native encoding file to an ascii file that includes the \udddd Unicode notation.

- Java RMI Stub Converter (rmic)
  Generates objects from the names of compiled Java classes that contain remote object implementations.

- Java Remote Object Registry (rmiregistry)
  Creates and starts a remote object registry on the specified port of the current host.

- Serial Version Command (serialver)
  Returns the serialVersionUID for one or more classes in a form suitable for copying into an evolving class.

- AWT 1.1 Conversion Tool (updateAWT)
  Included with the JDK AWT documentation, rather than in the bin directory. Updates deprecated 1.0 AWT names to new 1.1 AWT names (for Sun Solaris and UNIX systems, or Windows systems with the MKS toolkit).

- Various C libraries and include files

- Java Core Classes (classes.zip)
  This file contains all of the compiled .class files for the JDK.

- Java Source Files for Public Classes
  (src.zip file or src directory)
  This is the set of source files used to create the classes included in the Java Core Classes classes.zip file (above).

APPENDIX D

The source code and compiled byte code for the desktop manager and utility programs are included on a compact disk inside the back cover of this thesis.

VITA

Scott Griswold received a Bachelor of Science degree in Electrical Engineering from Tufts University in 1972. Scott expects to receive a Master of Science in Computer and Information Sciences from the University of North Florida in May of 1998.

Scott has held various design, field installation, and field service engineering positions since graduating from Tufts. He worked for over eight years exploring for oil and gas in Louisiana and Texas before moving to Florida. While in Florida he gained several years experience in the paper and process industry, and over ten years experience in electrical generation. Scott's interests lie in computer applications that aid in manufacturing and process control.