Cross Platform Issues in Software Design and Development: A Case Study of AthenaMuse 2

by

Issam Bazzi

B.E., American University of Beirut (1993)

Submitted to the Department of Civil and Environmental Engineering in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

Author .....
Department of Civil and Environmental Engineering
May 2, 1997

Certified by .....
Steven R. Lerman
Professor of Civil and Environmental Engineering
Thesis Supervisor

Accepted by .....
Prof. Joseph M. Sussman
Chairman, Departmental Committee on Graduate Studies

Cross Platform Issues in Software Design and Development: A Case Study of AthenaMuse 2

by

Issam Bazzi

Submitted to the Department of Civil and Environmental Engineering on May 9, 1997, in partial fulfillment of the requirements for the degree of Master of Science

Abstract

This thesis addresses the main aspects of cross platform software design and development. A software is cross platform if it can run on two or more platforms and provide the same logical functionality as well as a similar look and feel. A deep understanding of the required functionality and the capabilities of the different platforms to provide this functionality is very important in early stages of the cross platform software development process. We present different kinds of platform independence as well as the approaches used to achieve them through a case study of a platform independent, object-oriented, multimedia authoring environment, AthenaMuse 2. We highlight several innovative cross platform techniques that were developed in AthenaMuse 2 ranging from a platform independent user interface to abstraction layers for accessing and moving data in heterogeneous environments. A comparison with the Java programming environment, as an alternative solution, is also presented.

Thesis Supervisor: Steven R. Lerman
Title: Professor of Civil and Environmental Engineering

Acknowledgments

To my advisor, Prof. Steve Lerman, for his guidance and input during the writing of this thesis and for his support at CECI.

To Dr. Jud Harward, and all the members of the AthenaMuse development team, for their support during the development of AM2.

To CECI graduate students who always helped with their suggestions and encouragement. Among them, Richard Rabbat.

To everyone at the MIT Center for Educational Computing Initiatives.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's computing world depends heavily on complex and heterogeneous computer systems composed of personal computers, workstations, and interconnecting networks with a variety of operating systems and interface environments running on these systems. Computer users and organizations that rely on current information technology (IT) realize that every platform offers different benefits, and that these different platforms can be integrated with the advances made in network software compatibility.

As a result, users are looking for software to standardize across the different platforms. The need for a standardized software to execute on multiple platforms is not a new idea. Companies have been developing software with versions for every platform that will behave in a similar fashion across these platforms. FrameMaker™ and Lotus 1-2-3™, for instance, are good examples as they have versions that run on Windows, Macintosh and UNIX™ systems.

In most current software development practices, programmers are targeting their applications to multiple platforms. Initially, the approach was to develop separate applications with separate source code for each platform. In today's development environment, however, developing an application several times is not viable solution for companies that are competitive. A continuing effort is being made to help develop the tools and techniques necessary to create single source code applications that can be ported across different platforms[GLAD95].

1.1 Background

AthenaMuse 2™ (AM2) is a platform-independent multimedia authoring system that grew out of a research project at the Center for Educational Computing Initiatives (CECI) at MIT between the years of 1992 and 1996 [HRW94]. AM2 is best described as a multimedia authoring tool designed for authoring by multiple users in a heterogeneous, networked environment. At the heart of AM2 is a scripting language called the Application Description Language (ADL). ADL is the platform-independent storage format for AM2 application descriptions[AM2D97]. The original intention was to build a series of direct manipulation editors to assist users of all levels of programming experience in developing AthenaMuse applications. Only one such editor, a prototype layout editor, has been developed. Consequently, the ADL has also become the primary authoring medium for AM2 applications. The AthenaMuse environment currently runs on three flavors of UNIX™ (SunOS 4.2.n, Solaris 2.5, and HPP-UX 9) as well as on Win95 and Windows/NT 3.5.1. A preliminary version of AthenaMuse runs on Macintosh System 7, but as of the date of this document, this version is not supported.

This thesis attempts to highlight the main aspects of cross platform software design and development by presenting the cross platform strategies used in AM2. It also highlights some similarities and differences with the Java programming language, a new possible solution for developing network-centric, platform independent applications.

1.2 Goals of Cross Platform Development

Cross platform development evolved from portable coding practices[PETR94]. Portable source code will compile and execute on another computer system even if the underlying hardware is different.

The problem that portability solves represents a subset of the problems that a cross platform solution must solve. Portability deals with differences in hardware and operating

system features, such as memory and file management. Many of these differences are masked by compiler specific data types, careful coding practices and use of standard libraries.

A cross platform solution must solve the portability problem and also the problems specific to platforms: graphical user interfaces, event-driven operating systems, resource management, drawing graphics, displaying images, rendering fonts, and interprocess communication. Here is a list of goals that a cross platform solution should meet:

Platform Look and Feel

Since every platform has its unique look and feel, it is important that application takes on the look and feel of that environment. In some cases, application users might prefer a uniform look across platforms which may be different from all platforms looks.

Inter-Application Communication

Inter-application communication should be independent of the platform if the applications are on different platforms. This requires both a platform independent protocol for communication as well as a platform independent representation for data exchange. A good example is establishing an FTP (file transfer protocol) session from a Windows machine to a UNIX workstation where there is an FTP implementation for each platform.

Resource Exchange

A file or database record created on one platform should be readable and modifiable from other platforms. The ability of an application to do such task will require knowledge of platform specific data types and byte ordering. This knowledge is usually part of some underlying layer and is hidden from the application. For example the ability to read and modify a word processor file written on Windows from a Macintosh system is a necessary requirement of a cross platform solution.

Data Handling

An application should be able to process and interpret the same data in a similar fashion on all the platforms it runs one. One example is the ability to play a clip of digital video on the platforms of interest using the same data for the video clip.

We will go in more details on several kinds of cross platform software when we discuss the types of platform independence in chapter 3.

1.3 Benefits of Cross Platform Development

There are many benefits for cross platform development. The most important and obvious reason is the ability to run the same software on multiple platforms and to be able to move resources across them. Here is a list of some of the benefits:

Efficient Use of Software Development Resources

Producing a single application on a single platform requires knowledge of the that particular platform and the functional components of the application. Developing a single application on three platforms without using cross platform development techniques would require three different designs and implementation of the application on those three platforms. When using cross platform development, a single set of source code can be developed for all platforms, with some small part of the source code being platform specific. This cuts down the implementation size and time by a significant factor.

Easier Maintenance

The time needed to produce an application represents only a part of the cost associated with this software during its life. Continuous improvement and changes are always needed. If cross platform solution is adopted for developing such an application, there is a high probability that adding a feature or fixing a problem can be done once for all platforms. This is especially true for adding new features if system specific functionalities

have been abstracted into a set of high level objects or more simply to a group of functions that are independent of the platform they are on. Typically, a cross platform solution consists of a set of layers of abstraction. Each layer provides a certain level of abstraction for some specific purpose.

Supporting New Platforms Easily

Viable new platforms do not emerge often, and the long-term success of a new platform is often difficult to predict. Even successful platforms have a life cycle that is usually dictated by the success of a particular kind of computer hardware and the applications that are available for that system. Using a cross platform solution makes it easier to support new platforms and to quickly to move to a totally different platform environment with minimal design and implementation changes.

Same Look and Feel

One important benefit of cross platform software is that the software developer can make the look and feel of the application the same on all the platforms it runs on. A user using an application on Windows, for example, will become accustomed to the specific look and feel of that application. When the user moves to a different platform, he must learn the application interface if the look and feel are different. If the look and feel of the application are the same across platforms, the user can easily use the application. In large organization, and on a single user level, this removes the extra step of training to learn running the application on a different platform.

Market Advantage

Providing multiple platforms is always a marketing plus. It makes a software product visible and possibly more appealing to multiple-platform users. Providing a solution on one popular platform gives the software a lot of attention, but making it available on many

platforms gives the software the lead over the competition. This has been the case in most commercial products that were made available across platforms such as Lotus 1-2-3 and FrameMaker.

1.4 Organization of this Thesis

In this chapter, we gave a short introduction and background information about cross platform software design and development. We also listed the main goals and benefits of adapting a cross platform solution for software development. The rest of the thesis is organized as follows.

Chapter 2 provides a general overview of AM2, Java, and the various platforms they are implemented on. AM2 is described in more depth, with concentration on the cross platform aspects of this system. For comparison purposes, we give a short introduction to Java and how it relates to the work described in the thesis. We then present some of the properties of the three platforms that AM2 was developed on. We mainly focus on those details of the platforms that are relevant to this work.

In chapter 3, we present a more formal definition of cross platform development. We then detail the different kinds of platform independence that should be considered. Finally, we present various approaches to achieve independence that have been adopted in this area.

Chapter 4 goes into more details in describing how AM2 achieves different kinds of platform independence. Two major components of AM2 are presented, namely, the user interface (UI) and the media engine. Few useful mechanisms for achieving cross platform UI implementation are presented. At the end, a comparison is made between AM2 and Java event handling.

In chapter 5, we describe data management in AM2. First, we review the approach to implementing an abstraction of the file system. Then, we give a brief introduction to the AM2 virtual database module. Finally, we present a new approach for utilizing all kinds of data using a single, platform, independent generalized stream interface.

In the last chapter, we conclude with a list of lesson learned from this work and what the future holds for cross platform software development.

Chapter 2

General Overview

2.1 Introduction

Building cross platform software requires a deep understanding of how each platform the software will run on works and how different features can be made available on these platforms. In this chapter, we give a general overview of the three platforms that AM2 was built on, focusing mainly on those aspects that make these platforms different. Special attention is given to graphical user interface (GUI) since it usually constitutes the largest part of code that cannot be used across platforms. Three platforms are presented: UNIX running the X Window system, Microsoft Windows NT/95™, and the Macintosh system 7 platform. Following in the chapter is an introduction to the general design of AM2 and a brief introduction to the Java programming language and how it compares to AM2.

This chapter should provide a general background necessary for the rest of this thesis. For more detailed information, a list of references is provided at the end.

2.2 Overview of Platforms

AM2 was developed on several platforms which can be grouped into three categories:

- UNIX: this includes different flavors of UNIX running on different architectures, for example: SUN, HP, and DEC.
- Microsoft Windows: for example Windows NT, 95, and 3.1; all running on the Intel architecture.
- Macintosh OS: this is Mac operating system with MacApp as the framework for building applications.

The following three sections present an overview of the three platforms.

2.2.1 The UNIX Platform

One thing special about the UNIX platform is that it was not intended to run on one specific type hardware. Some of the platforms that the UNIX operating system runs on include PC's, SUN Workstations, IBM RS/6000 workstations, and DEC computer systems. Over the past 20 years, UNIX had many versions starting from versions 1-5 to BSD versions to Mach and AIX. This diversity of architectures spawned different looks of the UNIX operating system such as Motif and OPEN LOOK with the X Window system [GLAD95].

Main features of current UNIX operating systems are: multiuser support, multithreading, reliable security, network connectivity, and hardware independence through device drivers. But most of all the existence of a large set of standard UNIX utilities that makes the system easy to use when moving from one hardware type to another (Fig 2-1).

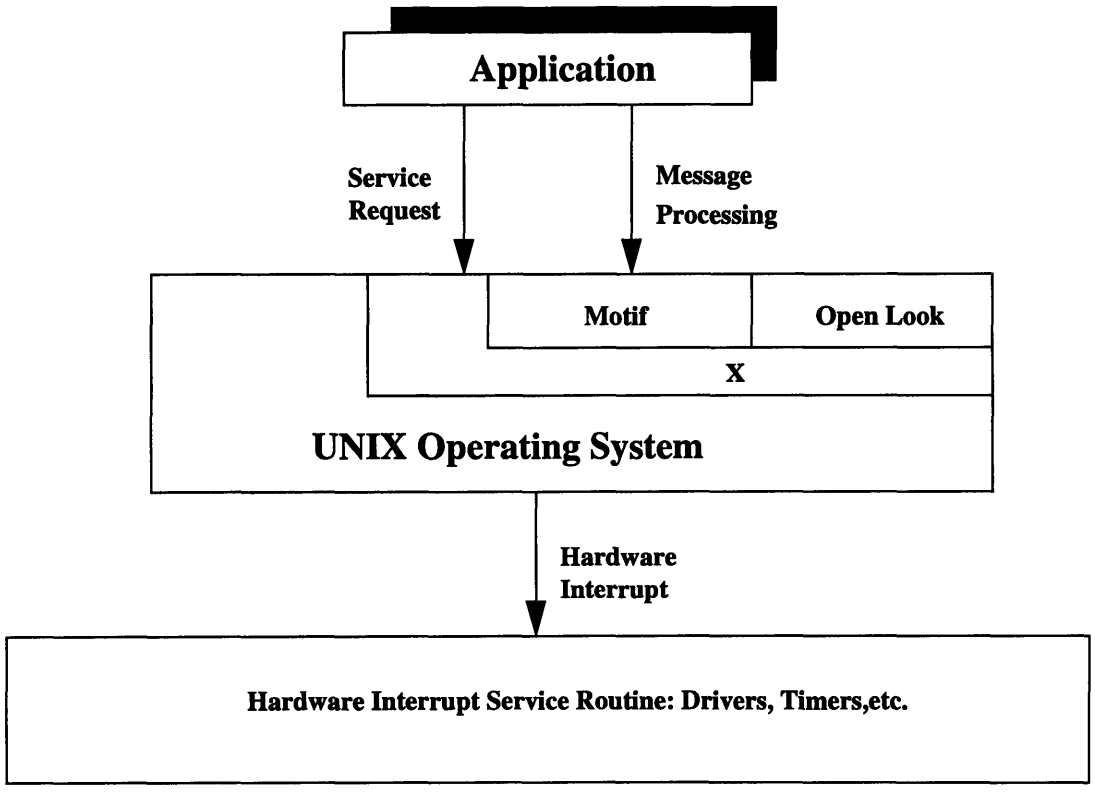

Figure 2.1: UNIX operating system [GLAD95]

A basic UNIX system provides a text mode only interface for the user. For GUI, Motif is used with support by the XWindow system. X is a hardware independent, graphical windowing system that controls a "bitmapped" display that allows applications to draw pictures as well as text.

UNIX Input/Output

As mentioned above, a user can interact with the operating system either in text mode or in a GUI mode using X. We are more interested in the second case for a multimedia application. Here we give a small example.

Figure 2.2 illustrates an example of the interaction between a GUI application and the operating system. When the application starts, it creates a window using a call to XtManageChild function. This is then propagated to a function in the X Library.

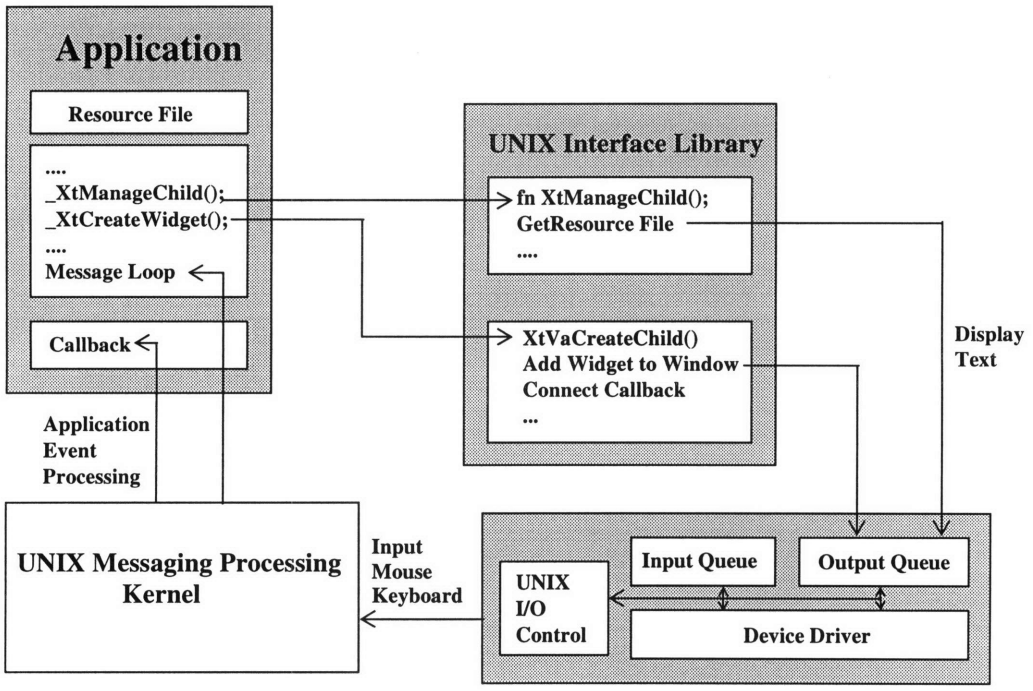

Figure 2.2: UNIX X-Windows Application Interactions[GLAD95]

The XtVaCreateManagedWidget reads the resources for the window from the resource file supplied by the application and then displays the window on the screen. The operating system then takes care of all messaging and interaction between the user and application. Usually, every interaction is associated with a callback function. The overall effect of the callback mechanism makes an X application event-driven [GLAD95].

The development of different versions of UNIX brought some problems for UNIX developers since they need not only be concerned with portability to other operating systems but also need to be concerned with porting among the UNIX platforms. A standardized version of UNIX has been proposed by IEEE by developing the Portable Operating System Interface based on UNIX (POSIX). The idea is that applications written with this interface should work on all platforms[ISAA94]. Some non-UNIX operating systems took advantage of this. For example, Windows NT supports POSIX, which allows for easy porting of UNIX application to Windows NT.

2.2.2 Microsoft Windows

Even though there are many flavors of Windows, we will focus here on Windows NT for two reasons: first, AM2 was developed on Windows NT, second, the basic features of Windows NT cover all other flavors of Windows (95, 3.1, 3.11 for Workgroup).

Windows NT Architecture

NT borrows from the better points of the other systems, and adds an object based view of operating system components and all of the objects it manipulates. The NT architecture is distinct in that it employs the following properties[BRAI94]:

Hardware Abstraction Layer:

HAL is virtualized machine architecture that represents the true underlying hardware, for example: the Floating Point Unit, CPU registers, Virtual Memory Management Unit,

etc.The operating system is made largely independent of the supporting hardware. This simplifies porting NT to different platforms. For each platform, the HAL is replaced.

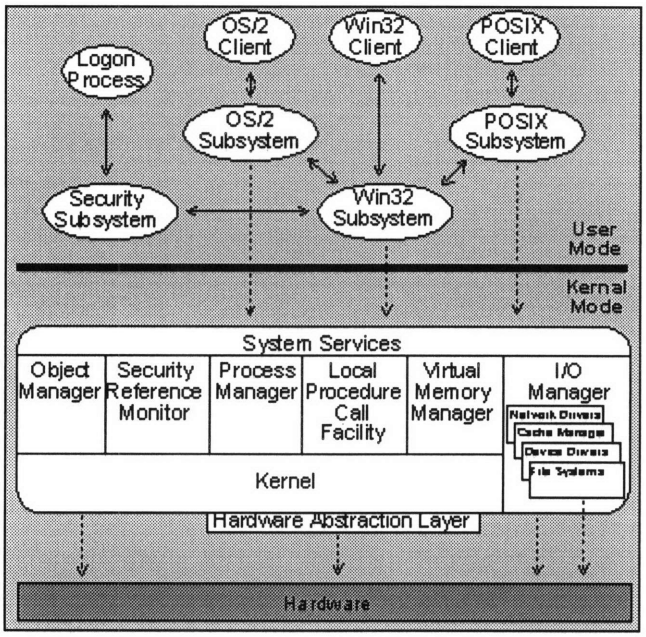

Figure 2.3: NT Operating System Architecture

Object Based Model:

NT itself is modular and object based, and its view of all entities is also object based, i.e. memory, files, users, devices, etc.

Message Passing:

Many subsystems communicate by passing messages, instead of traditional subroutine calls. This highly decouples dependency on specific subroutine interfaces.

User-Mode Interface Subsystems:

Recognizing that the interface may be de-coupled from the operating system proper, NT has User-Mode Interface subsystems. NT may support any number of user interfaces, and new ones may be added quickly.

NT Input/Output

An NT application interacts with the operating system in a similar fashion to UNIX. When a GUI application starts, a resource file is used to decide how and what kind of windows and controls should be placed on the screen. This resource file is usually linked into the application binaries when the application is compiled or stored in a Dynamically Linked Library (DLL) that allows for run time linking of routines and resources[EZZE93].

All GUI calls are passed to a user interface library that takes care of managing different windows (See Fig 2.4). In some cases the application might be built using a framework such as Microsoft Foundation Classes (MFC) which provides a fully Object Oriented Interface to the windowing system.

All messages from and to the user are handled via an application-global dispatcher that takes care of passing the messages to the right window handler called *Window Procedure*. In Windows, messages can be intercepted at any level and an alternative or an additional action can be taken. The process is known as subclassing a window. This can be done more than once for a single window, thus creating a chain of window procedures or handlers where the latest handler receives the messages first, decides what to do and whether to pass them to the previous one, and so on.

Dynamic Linking in NT

Dynamic linking provides a mechanism to link applications to libraries at run-time. In contrast to a static library, the code in a DLL isn't included in the executable file that uses the DLL. Instead, a DLL's code and resources are in a separate file (usually with a .DLL extension). This file must be either currently loaded in memory, or accessible to windows when the client program runs.

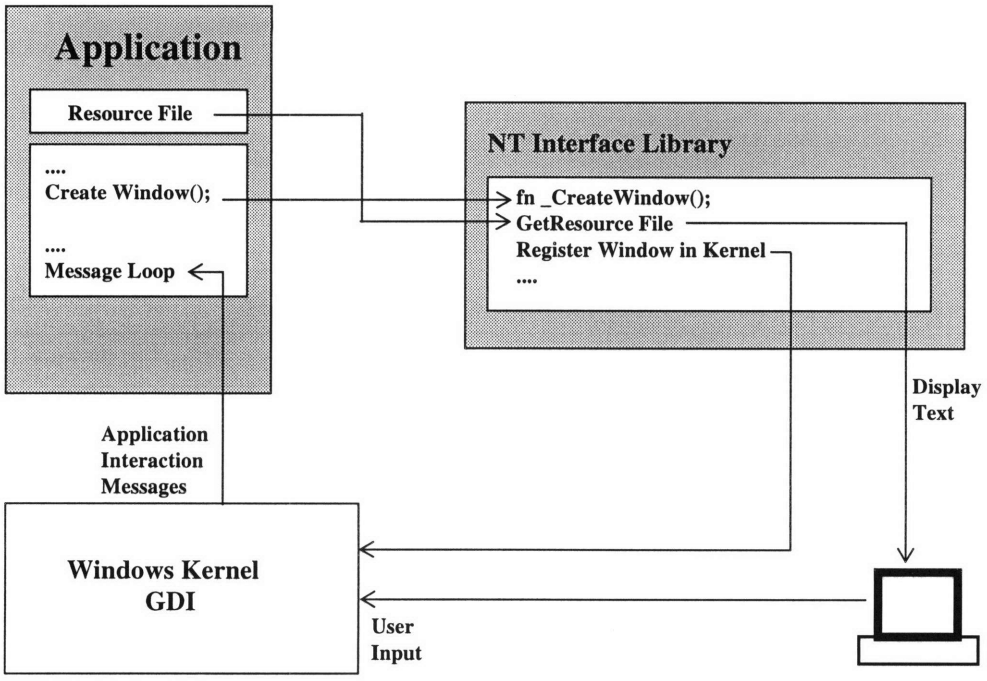

Figure 2.4: NT Application Interaction [GLAD95]

When Windows loads a DLL or an executable file into memory, it scans a list of all the names of DLL's which required to execute the application. Any DLL needed gets loaded into memory at the same time if it is not already present. Alternatively, using the Windows API, the function call LoadLibrary() allows loading a DLL into memory only when it is actually needed, and unload it when it is through.

When a DLL is loaded into memory by the operating system, its procedures are accessible by all other programs (or DLL's). Only one copy of the DLL needs to be present in memory. This is possible because the library is not linked into any one of the programs permanently. It is just present, in memory, making its services available to any code that may need them. UNIX has a similar capability through the use of shared libraries.

2.2.3 Macintosh

We give here a brief overview of System 7. Figure 2.5 shows the general design of the Mac operating system. Many of the graphical elements were embedded into the hardware which allows for fast and sophisticated graphical user interface applications on the Mac.

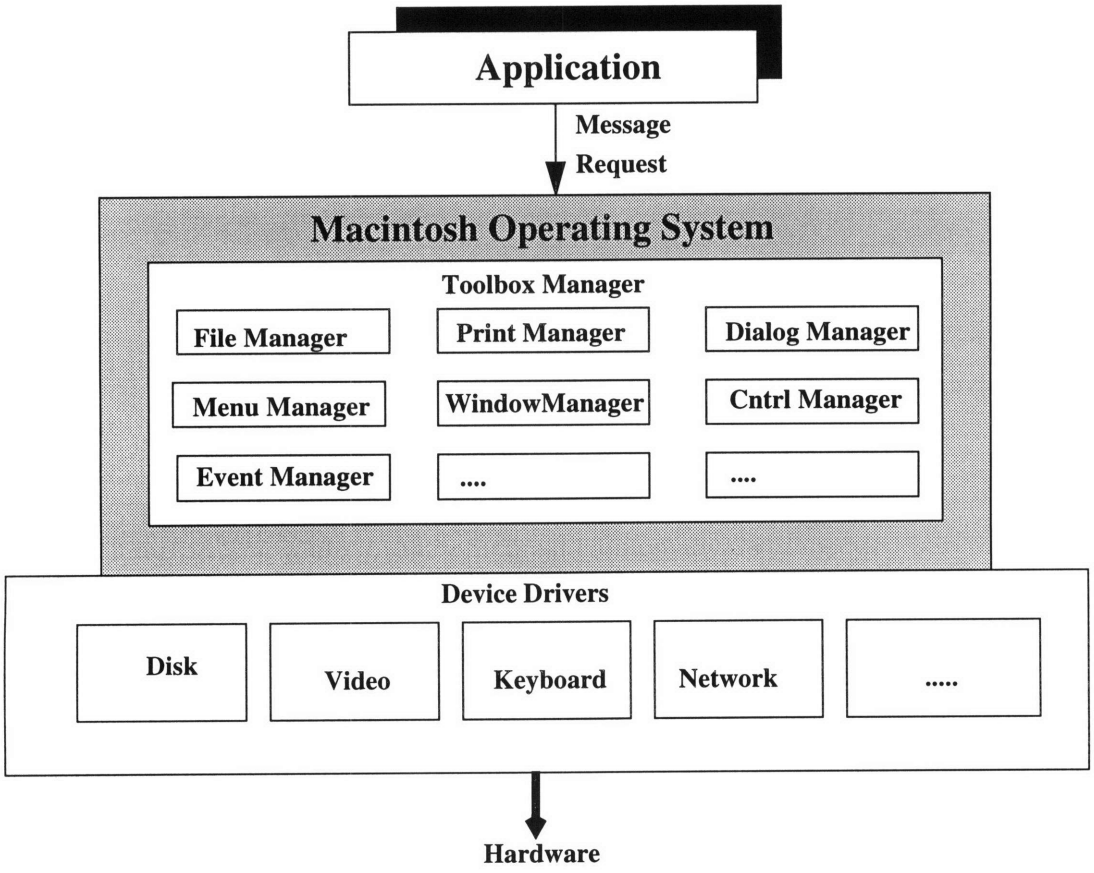

Figure 2.5: Architecture of the Mac Operating System

The operating system is an event-driven, pre-emptive OS with all events processed by the Toolbox event manager with some priority scheme for events. There are different managers for operating system services. Some of the managers are shown in Figure 2.5.

Mac input/Output

The interaction between a Mac application and the operating system is event-driven.

When the application starts, it requests a window creation which propagates to the Toolbox interface library. The call in turn requests information from a resource file and places the window on the screen.

Events and notifications are handled by an event manager, part of the Toolbox subsystem as shown in Figure 2.6, in a fashion similar to UNIX and NT. What makes the Macintosh interface unique is that most of the operating system is located in the hardware[GLAD95].

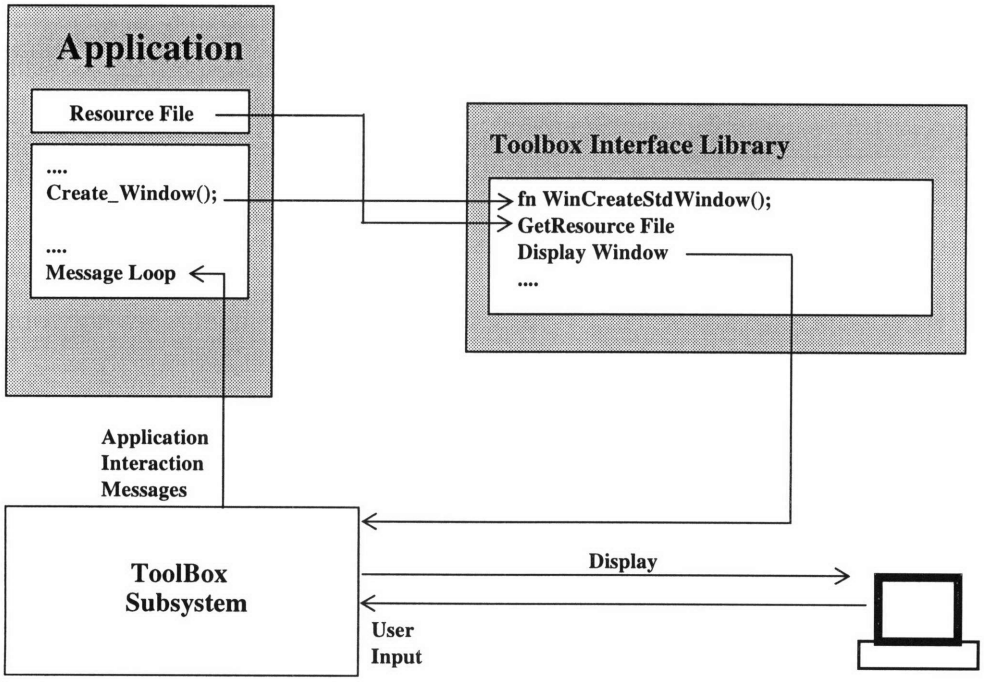

Figure 2.6: Mac Application IO processing [GLAD95]

MacApp/OpenDoc

In many cases instead of writing an application using the operating system routines directly, a framework is used. For Macintosh, the most common is MacApp. MacApp is an object-oriented framework and class library used for building various kinds of Macintosh applications. It streamlines development by supplying the application main-event

loop and code for all basic Macintosh features. MacApp is a C++ class library. The library implements many of the visual and organizational elements of most Macintosh programs. As a framework, MacApp provides the foundation for a standard way to write Macintosh programs. Many major applications have been written in MacApp. AM2 on Macintosh was developed in MacApp.

A more recent and more interesting Mac framework is OpenDoc. The OpenDoc Development Framework, or ODF, is an object-oriented framework developed in C++ which is targeted for building cross-platform OpenDoc component editors. Like MacApp, Apple's framework for building stand-alone Macintosh applications, ODF makes the process of building an OpenDoc component editor easier by implementing much of a component editor's default behavior.

OpenDoc is supposed to be delivered on a variety of platforms and provides a good foundation for building cross-platform component editors. On top of OpenDoc, ODF provides a set of cross-platform services to ease the development of OpenDoc component editors. One of the services needed to write a truly cross-platform component editor is a cross-platform graphics engine.

Other services include menu support and cross platform resources. The OpenDoc Development Framework builds on the technology provided by OpenDoc and cross-platform technology developed at Apple to provide a complete cross-platform solution. ODF should allow a developer to develop component editors once and have them work on multiple platforms.

2.3 AthenaMuse 2 General Overview

This section gives an overview of the design goals of AM2, gives the application's general components and presents the main architecture of the system.

AM2 Design Goals

AM2 has several design goals that have affected the design of the system itself as well as application building environment it provides. These goals can be summarized as:

- form a flexible and extensible multimedia environment.
- support diverse media types and networking technologies.
- provide transparent application portability across X/UNIX, Macintosh, and Windows platforms.
- provide an easy-to-use, flexible, and complete set of multimedia application editors.

To achieve these goals, the description of an application's interface should be separate from the content presented. For example, in a multimedia application that contains a video viewer you may want to use the viewer many times, but each video clip viewed with it is tied to the particular context.

An application should be as portable as possible across platforms and environments so that you can customize an application to suit a user's background and preferences, and so that you can take advantage of special features of a particular hardware configuration or operating system.

For example, an application may use the English language on interface controls as a default, but it should also allow customization of the control labels in other languages. And the application should request general services, such as a video source, and determine how to access that source from a description of the system configuration.

Application Structure

Satisfying these goals suggests that an application consists of three distinct parts:

- The application description specifies the application's interface and functionality as pure and platform-independent a form as possible.

- Application content is stored separately from the application description.
- Customizations of the application are stored separately so that the same application description can run with different sets of customizations (These customizations in AM2 are called assets)

In AM2, classes describe the application's interfaces and functionality. At run-time, instances of these classes are populated with content drawn from databases, files, network services, or the application itself. The use of classes to specify interfaces encourages users to think in terms of, and to build with, nested interface templates.

AM2 Runtime Environment

Application Programming Interfaces (APIs) are necessary to facilitate communication and transfer of information among the architectural layers of *AM2*. Three APIs can be distinguished in AM2. These include the ADL, the Multimedia Toolkit (or MMTK) API, and the Device Independent (or DIX) API as shown in Figure 2.7.

The *AM2* runtime environment runs as an automaton composed of objects, many of which are instanced dynamically. In order to ensure the portability of the *AM2* environment and to simplify the addition of new media types, there are four categories of services below a device independent API (the *DIX API)*:

- Operating System Services
- Media Services
- User Interface Services
- Database Services

The *AM2* runtime environment makes requests for such services not from platform dependent libraries or device dependent modules, but from device and platform independent *AM2* objects residing in or above the *DIX API*. For example, all calls to X Window Sys-

tem routines or the UNIX file system are segregated below the *DIX API*.

This is illustrated in the figure below.

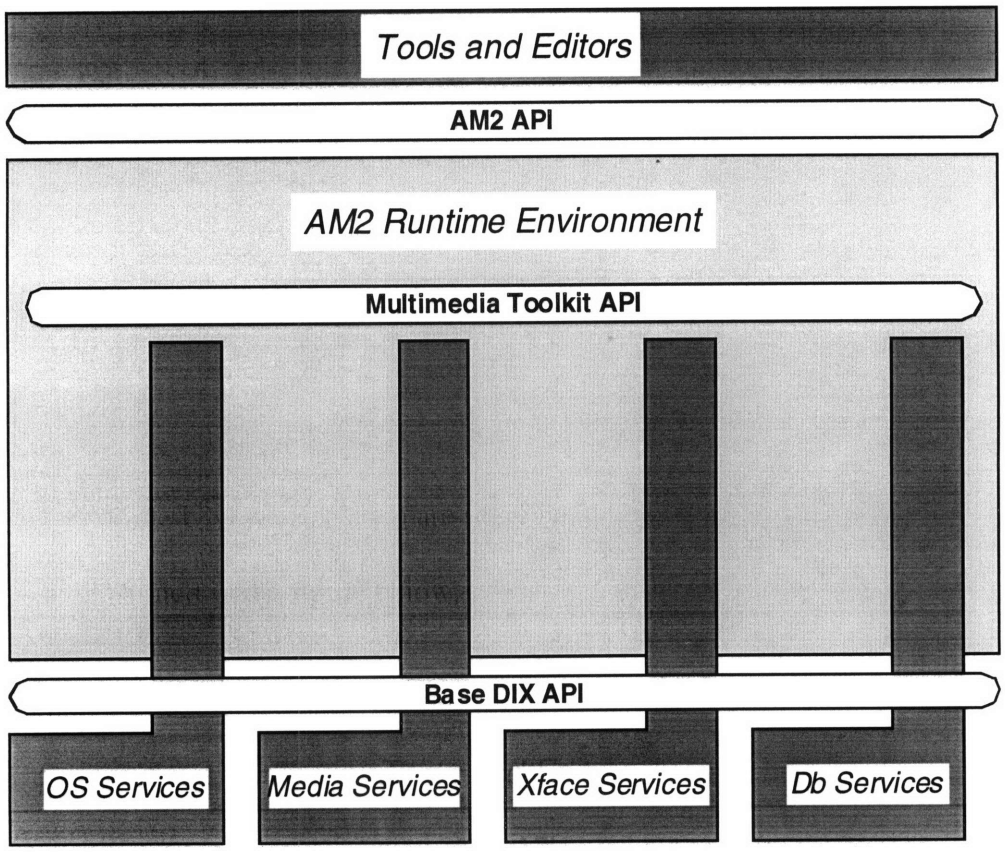

Figure 2.7: Application Programming Interface [AM2D94].

AM2 Modules

The core of the *AM2* runtime environment lies in the *control engine*. In very simple terms, a *user interface module* messages objects in the control engine (*control objects)* to notify the automaton of user-generated events, such as button presses. The control engine updates its own state appropriately and alters the application's display by messaging the appropriate objects in the *media engine, user interface module,* and *database servers.* In more detail, we will segregate functionality as follows:

Control objects: manage the application event loop and process callbacks, supply application content to interface objects, maintain application state. Control objects can be created dynamically to support application development tools and the incremental parsing of application description files by a separate parser object.

Media objects: provide access to application content and implement the display of that content via media object methods. This allows control objects to call for the display of media without knowing the underlying media type. Media objects manage the device dependent details of each media type as well as device contention and tight media synchronization at the level, say, of combining an audio stream with an independent video stream.

User Interface objects: (UI objects) manage the windows and widgets of the underlying windowing system in a platform independent fashion. These user interface objects are created in response to messages from control objects, which coordinate the association of UI and media objects. UI objects also message control objects in response to user input and other window system events.

Database objects: provide application content on demand to the media and control objects that request it from different kinds of databases.

2.4 Introducing Java

The main goal of the Java language was to develop applications in the context of heterogeneous, network-wide, distributed environments. Among many challenges to this goal is the secure delivery of applications that consume the minimum of system resources, run on any hardware and software platform, and have the ability to be dynamically extended.

Design Goals of the Java Language

A primary goal of the Java language is a simple language that could be programmed without extensive programmer training and which would be roughly attuned to current software practices.

The Java language is also intended for creating highly reliable software. Emphasis is on extensive compile-time checking, and a second level of run-time checking.

The Java language was designed to support applications executing in networked environments, operating on a variety of hardware architectures, and running a variety of operating systems and language environments. The Java language compiler generates byte codes--an architecture-neutral, intermediate format used to transport code to multiple hardware and software platforms.

In addition, the Java language supports multithreading at the language level with the addition of some synchronization primitives, at the language library level, and at the run-time level with monitor and condition lock primitives.

While the Java compiler is strict in its compile-time static checking, the language and run-time system are dynamic in their linking stages. Classes are linked as needed. New code modules can be linked in on demand from a variety of sources, even across a network.

The Java Virtual Machine

The Java Virtual Machine (VM) is the software implementation of a "CPU" designed to run compiled Java code. This includes stand-alone Java applications, as well as "applets" that are downloaded and run in Web browsers such as the Netscape Navigator.

In other words, the Java VM is the part responsible for Java's cross-platform delivery, the execution of its compiled code, and its security capabilities. The Java virtual machine consists of:

- An instruction set
- A set of registers
- A stack
- A garbage-collected heap
- A method area

All of these are logical, abstract components of the virtual machine that allow for full platform independence. They do not presuppose any particular implementation technology or organization, but their functionality must be supplied in some fashion in every Java system based on this VM. The Java virtual machine may be implemented using any of the conventional techniques: e.g. bytecode interpretation, compilation to native code, or silicon. The memory areas of the Java virtual machine do not presuppose any particular locations in memory or locations with respect to one another. The memory areas need not consist of contiguous memory. However, the instruction set, registers, and memory areas are required to represent values of certain minimum logical widths (e.g. the Java stack is 32 bits wide).

Limitations of Java

Part of the notion behind Java is to simplify the programming model of C++ and thereby prevent common programming faults, such as improper uses of pointers. Therefore, the Java VM does not provide access to them. This limitation has been a source of serious trouble to Java developers as pointers usually allow for a more flexible and dynamic memory usage.

Java's VM supports only single inheritance. Single inheritance was chosen to simplify the programming model (and maybe the VM implementation). But in many cases, the use of multiple inheritance can create clean and reasonable class structures.

Another problem with Java is the lack of object persistence. Persistence implies that memory management involves both in-memory and on-disk objects. Java hopes to glue on interfaces to relational and object databases in order to obtain a capability important to virtually all professional level applications.

Other problems with Java come from the fact that the performance is not as good as compiled code and the language is still evolving with major changes from one version to the next. In addition to the fact that learning the language is not as simple as it is supposed to be.

2.5 Conclusion

In this chapter, we gave an overview of the three platforms that AM2 was built on. Even though the design of the three operating system is some how similar, many challenges face software developers when trying to develop cross platform software. AM2 is one good example. The goals of AM2 were to meet these challenges by providing a platform independent environment. Coming from a net-centric point of view, Java tries to solve the platform problem in addition to providing robust and secure communication among many other features. In the following chapters, we go in more details into how platform independence can be achieved in general and in AM2 and Java in particular.

Chapter 3

Cross Platform Software

This chapter presents different kinds of platform independence that application developers try to achieve when developing a cross platform solution. It also presents some of the most common approaches for building such applications. The chapter starts with some definitions for different components of a cross platform system.

3.1 Definitions

A general cross platform system can be described by Figure 3.1. The figure shows two platforms and three applications, one for each and a cross platform one that can run on both. In this section, we define the different components shown that constitute the main building blocks for different kinds of cross platform development[PETR94].

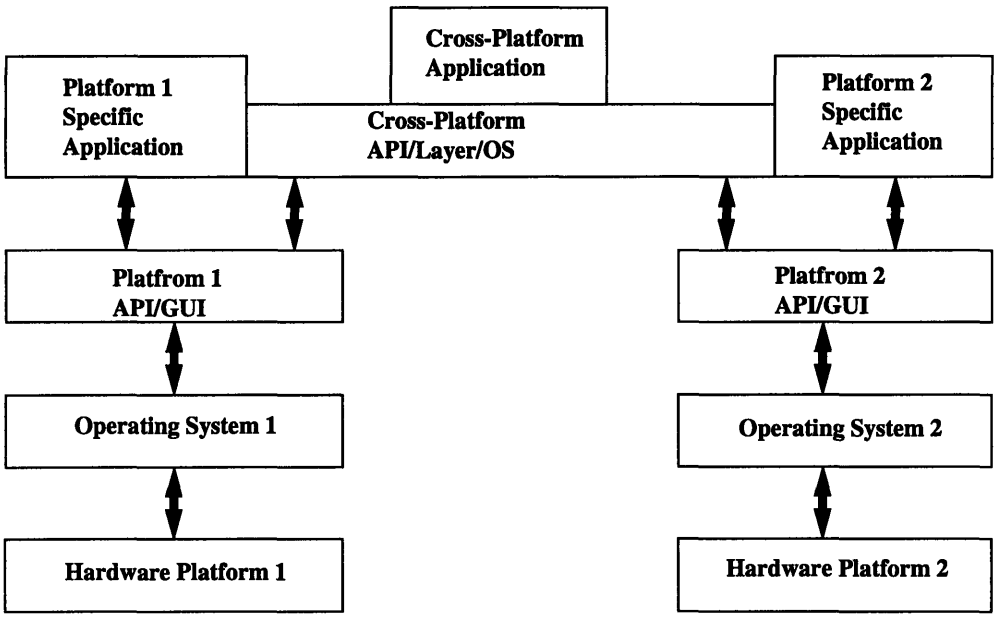

Figure 3.1: A General Cross Platform Architecture

Hardware Platform:

A hardware platform is a computer hardware design that incorporates a certain type of microprocessor; for example, a PC, which is based on the 80X86 microprocessor, is a hardware platform and a Macintosh, which is based either on the 680X0 microprocessor or the Power PC architecture, is another hardware platform.

A hardware platform is capable of running any type of operating system. Some hardware platforms run only one type, while others run more than one type; for example, the Macintosh runs System 6, System 7, or A/UX; the PC runs DOS, Windows NT, OS/2, UNIX, NeXTSTEP, Solaris, etc.

Graphical User Interface (GUI):

A GUI lets the user access software through a graphical (usually non-textual) paradigm. Typically a GUI uses icons to represent an application or documents. An application usually maintains one or more windows, a menu bar, and dialog boxes with the user will interact.

Typically the user positions a mouse cursor over an icon and double-clicks to start up the application or document. Other methods of interaction include clicking and dragging files to copy or move them from one directory to another. A platform's GUI is unique but still shares many anatomical features of most other GUIs. The Macintosh, Windows 3.1, Win32 (Windows NT), Presentation Manager (OS/2), and Motif (X) are a few of the most popular GUIs.

Operating System:

An operating system is a program that runs very close to the computer hardware (usually just above the ROM code or BIOS). It gives the user a way to access system and file information and run applications. An operating system also provides the developer with a

set of access points which a application uses to obtain operating system information or perform low-level tasks, like reading the contents of a file. The user interacts with the operating system using a text-based command line (as in a UNIX shell) or using a GUI as in Windows NT.

Application Programming Interface (API):

The API is a set of functions or routines that the application developer uses to access the features and capabilities of the system or an underlying library. One example is a GUI API that allows the control of user interface elements.

Platform-Dependent or Platform-Specific Feature or Code:

A platform-dependent or platform-specific feature is unique to that platform (for example, a single desktop menu bar is unique to the Macintosh). Platform-dependent or platform-specific pieces of source code will run only on a specific platform.

Platform-Independent Feature or Code:

A platform-independent feature is independent of the platform it will run on. A platform -independent piece of source code can be compiled to run on any platform. When an entire application is produced in a platform-independent way, it is a cross-platform application.

Resource:

A resource is data that is bound to an application and is necessary for that application to run. For example, an application's resources might define a new cursor shape, an icon, or the contents of a dialog box. In the Macintosh a resource can be modified when an application is running. In Windows NT a resource is usually statically bound to the executable file during the development process and is not modifiable at run-time.

3.2 Types of Platform Independence

Platform independence can be classified into several types. The most common of all is the user interface since it is usually the one that requires most of the work when moving an application onto a different platform. We divide platform independence into the following types:

- Machine/Human interaction
- File system access
- Database access
- 2D and 3D Graphics
- Network communication

In the following sections we discuss each of these types of platform independence.

3.2.1 Human/Machine Interaction

When developing an application the first step is to design, and possibly to prototype, the look and feel that the application will have. All targeted platforms must be considered at the beginning of the design effort. The look and feel will help to identify the actual programming interface and structure the application requires to accomplish this specific look and feel.

A user interacts with the computer through a set of hardware and software components. The hardware components consist of the monitor for output and either a keyboard and/or a mouse for input and interaction with the user. The software components can be placed into three separate functional groupings. These groupings include high level functions, allowing the application to display output in the form of objects; the low level functions, providing the application with direct output for the intrinsic graphic hardware; and

the platform-specific functions, providing the actual hardware interface to these hardware devices.

On the vast majority of currently available platforms, a user interacting with an application can use one of two different types of input devices. These devices are the keyboard and the mouse. For software interfacing, the keyboard is the simplest device. Information is typed into the computer from the keyboard. The keyboard input can be either character mode or line mode input. DOS, for example works only in text character mode, while UNIX can work either in line or in character mode, depending upon the settings done in the terminal. In UNIX text line modes, software drivers interact with the keyboard, keeping all input in a buffer until a terminating character is entered.

3.2.2 File System Access

An application accessing data from files stored on the disk can be different from platform to platform. The primary difference between applications is the identifier of the files being accessed (file name). Table 3-1 provides a summary of the differences in file name sizes for the relevant platforms. The table illustrates significant differences between the lengths of the file names in the platforms. Applications planned for multiple platforms need to be consistent in the naming convention for the file names across the platforms. If the application is relying on the native file routines for accessing data, a platform-specific code is needed to process each platform's data. The DOS file names are a standard 11 characters with 8 characters names and 3 character extensions. This file naming convention is the same in Windows since Windows runs as a user interface on top of DOS. The Windows NT system supports the DOS file names, and a larger file naming convention of its own high performance file system (NTFS) with file names being 256 characters. Macintosh file names are 31 characters long with or without extensions. The UNIX system is even more flexible with three formats. The first is the standard UNIX format with 14 character file

names. The second is the Berkeley System Extensions (BSD) to UNIX which allows for file names as long as 255 characters. The third is the Network File System (NFS) which also allows for file names as large as 1024 characters. The NFS is a portable platform due to its network application. NFS extensions allow applications running on a DOS machine to make requests for the NFS file system[GLAD95].

Other differences in file naming are case sensitivity and characters allowed in the file name. In Windows, for example, file names are not case sensitive while on UNIX and Macintosh they are. For systems that use pathnames, a slash is not allowed, while on Macintosh, they are allowed. Moreover, in earlier versions of windows blank spaces were not allowed.

Platform	File Name Length	Example
DOS	**11 characters** **8 character name with** **3 character extension**	**dos_exam.fil**
Windows **Windows NT**	**Same as DOS** **Same as DOS** **NTFS 256 characters**	**FILE_can_be_256_ch**
UNIX	**14 characters long** **BSD 255 characters** **NFS 1024 characters**	**FILE1234567890** **FILE1234567890_more** **....**
Macintosh	**31 characters**	**mac_file_31_ch_long**

Table 3.1: File Naming Across Platforms

3.2.3 Database Access

There are two different approaches for developing an application for multiple platforms that access specific databases. The first option is to use the database system API to develop the application. This option works only if the database system operates in all platforms that are targeted, since the application will be written in the database language and

will be executed in the database program itself.

The second option is to develop a C language program, and either develop an interface to the database or purchase an existing commercial interface that provides access to the specific database. These routines will be as complicated as indexing, searching, and retrieving records in multiple database files. The more complicated the access becomes, the more important it is to use a commercial interface that will provide all required functionality.

Some of the commercial database interface packages come with interfaces for a few different platforms. The reason for using one of these commercial libraries is to stay compatible with the data, which resides in some standard database format. One of the problems with using one of these database libraries is that these libraries are extremely large.

3.2.4 2D and 3D Graphics

The main problem with displaying graphical data on a variety of platforms is that every platform handles the graphical interface through its own set of display coordinates and display parameters. To take a formatted graphical file and display that file requires a set of interface routines that will interact directly with the targeted platform. Developers usually build a set of interface routines that will interpret the graphics file and output the image in the format that is supported by the platform's interface to the graphics routines.

3.2.5 Network Communications

The network interface is slightly different from all the other software interfaces. To implement an application that can communicate with applications running on a variety of platforms requires selecting the protocol that will be used in communicating with the network. The most widely used types of network communication is the TCP/IP protocol. When a protocol is selected, the developer must determine whether that protocol is supported by

every platform. For example, the TCP/IP protocol is supported by all platforms, but additional interface software is needed to provide that network interface link to the application. Supporting a common protocol and taking care of details such as the byte ordering in data representation allows for platform independent ways of communication between applications residing on different platforms.

The communications interface for an application provides the user with control of the communications devices. Every platform normally provides its own control for the communication access. For example, Windows handles the communications devices internally and buffers information for the application. The application does not directly interact with the device. UNIX controls the communications ports by treating them as streams. The Macintosh controls the communications ports in a similar manner to UNIX, where the application interfaces directly to the communication ports.

When developing an application, directly communicating with the hardware to input and output data would be consistent across the different platforms. All the application would need to do to execute the correct input/output statements for the platform.

If the data streams being received or being transmitted require a specific protocol, such as Kermit or XModem, then the developer has two choices in implementing this communications. First, the developer can create a custom interface providing the protocol to the application, or second, the developer can acquire a third-party library that will provide this protocol and interface the application to the communications hardware.

3.3 Approaches to Development of Cross Platform Software

There are numerous methods and techniques that are used in cross-platform development. Some of these methods are straightforward and don't affect performance but will limit the complexity of the resulting application. Others are more complex and may ultimately

affect performance but imply no limit to the application's final complexity. As with most solutions, there is a trade-off between the time needed to implement the solution and its usability.

Some techniques include a combination of the methods described below, mainly: porting an API to a different platform, functional abstraction, emulating functionalities from one platform to the other, data abstraction, and the uses of toolkits and object class libraries supported across different environments.

3.3.1 Ported API

Probably the most common approach to cross-platform development is finding a common denominator and then implementing an API that uses this common denominator and adds to it. With this approach, the developer analyzes the target platforms and their APIs to determine a subset of features that the application requires. Once the feature set is defined, the developer can design a new API or layer to remove existing platform dependencies. The design of this cross-platform API is supposed to produce applications that run on all supported platforms.

3.3.2 Functional Abstraction

Normally the common denominator method requires you to construct a new functional interface, or layer, that calls the corresponding function or functions in the underlying platform API. Ideally this functional abstraction is kept to a minimum to allow the cross-platform solution to take a fairly small development effort.

However, using an interface layer, adds an extra function call since one cannot call the platform routines directly. This is called an indirection of the platform function. An indirection of one function call rarely appreciably degrades performance on most platforms. A call to an abstracted function will indirectly call the native API function. In some instances

it isn't possible to find a functional equivalent on all platforms. When this happens, one needs to find the best fit for as many platforms as possible and then synthesize the functionality on the other platforms.

Creating, or synthesizing, functionality that is not present in a platform isn't a distinct cross-platform implementation technique. Rather, it is a way to augment another technique when a platform is deficient in one or more areas. For example, a cross-platform date and time function needs to synthesize date and time information by calling low-level functions. Unless all of the platforms for which you are developing a cross-platform API provide the same feature set, you will need to synthesize some functionality.

If the cross-platform API is too broad, it may be very difficult or even impossible to synthesize the required functionality across all platforms. For example, if some platforms support preemptive multitasking and others don't, it would be complicated to synthesize this functionality. In these cases it might be better to settle for less functionality. Because synthesis is very labor intensive, the decision to use this method really depends on how important a certain class of functionality is and how much time and how many resources should be spent to solve the problem.

3.3.3 Emulation

If a particular platform differs radically from other target platforms, emulating the source platform might be the easiest approach. This approach is a more encompassing version of rewriting and synthesis.

Emulation can also be decided ahead of time if the developer knows that there are so many differences among the platforms that it is better to start with a fully new API to be implemented and used across all platforms.

3.3.4 Abstracting Data

Data abstraction isn't a method unto itself, but is a necessary component of all the other methods. Data abstraction means that certain platform-dependent data types are abstracted or hidden, and sometimes enhanced, in order to produce a new (common) data type to be used by relevant platform-independent functions. Data abstraction is inherent in an object-oriented cross-platform solution and is part of an object class definition.

The required degree of abstraction depends on the complexity of the functions that will use the data type. For example, if one is developing a cross-platform solution for two platforms, Pa and Pb, with corresponding data types, Da and Db, it might be necessary to create a new data type Dc to produce a workable cross-platform functional interface. However, in other cases one might decide that the cross-platform data type should be equivalent to Da or Db.

Data abstraction allows the developer to specify additional data fields, modify data fields, or hide certain necessary (platform-dependent) data fields.

3.3.5 Using Object Class Libraries

An object class library is essentially the object-oriented equivalent of the common denominator method of cross-platform development coupled with data abstraction. One disadvantage of using an object-oriented language is the introduction of another level of indirection in the execution of an object method over the equivalent functional implementation. In non-demanding applications this creates an acceptable amount of overhead, but it can cause a marginally responsive application to become unacceptable or tedious for a user.

Many class libraries have been ported to other platform. Microsoft Foundation Classes (MFC) library is ported to the Macintosh by using a special cross compiler. AM2 provides

an object-oriented version of the GUI and the media services named the Multimedia Toolkit (MMTK) which we will describe in the next chapter.

3.4 Conclusion

In this chapter, we covered the main issues involved in developing cross platform software and discussed some common solutions. Determining which solution to choose depends on several factors. The first is whether the application has an existing implementation on one platform and an attempt is being made to move it to another platform. This will impose restrictions based on how the application behaves on the original platform. On the other hand, if the application is to be implemented on all platforms, any of the above listed solutions can be used. Second, the availability of some tools might make the porting process easier and might affect the choice of the appropriate solution. For instance, the existence of a cross compiler might allow for building the application on one platform and then cross compile it to other platforms. Finally, and the most important factor is the functionality required for the application. For example, if some sophisticated graphics is needed, it might be worth to develop a set of platform-independent and efficient interface routines to the operating system graphics engine.

Chapter 4

Platform Independence in the AM2 Multimedia Toolkit

AthenaMuse 2 is a multimedia authoring system that requests services from the windowing system through the use of a set of tools that can provide these services in a platform-independent way. This set of tools is called the multimedia toolkit (MMTK) and it is basically composed of two modules: The User Interface module (UI) and the Media engine module.The UI module consists of the User Interface classes which are the platform independent components that the Interface Control objects use to build application or module specific interfaces.The Media engine manages different media elements such as individual images, sound bites, film clips, text and other media units, which are included in an application.

This chapter describes the process of the design and development of this toolkit and focuses on the main cross platform issues involved in developing the MMTK. It also describes the implementation of the MMTK under Microsoft Windows NT and Microsoft Windows 95 and gives a brief comparison with Java.

4.1 Design of the Multimedia Toolkit

Being a part of a platform-independent system, the MMTK is designed in a way that fits the needs and plans of AM2. In general terms, the design of the MMTK takes into consideration a certain set of goals:

- platform independent media and user interface API.
- MMTK classes provide services to the control classes and ADL.
- C++ applications built with the MMTK do not require the control classes and ADL.

• generalized access to platform specific solutions.

The figure shown below (figure 4.1) shows the general architecture of the MMTK. There are three layers:

• Platform independent (DIX) layer.

• Device Dependent (DDX) public interface.

• Native system code (Win32 for Windows, MacApp for Macintosh, and Motif for Unix)

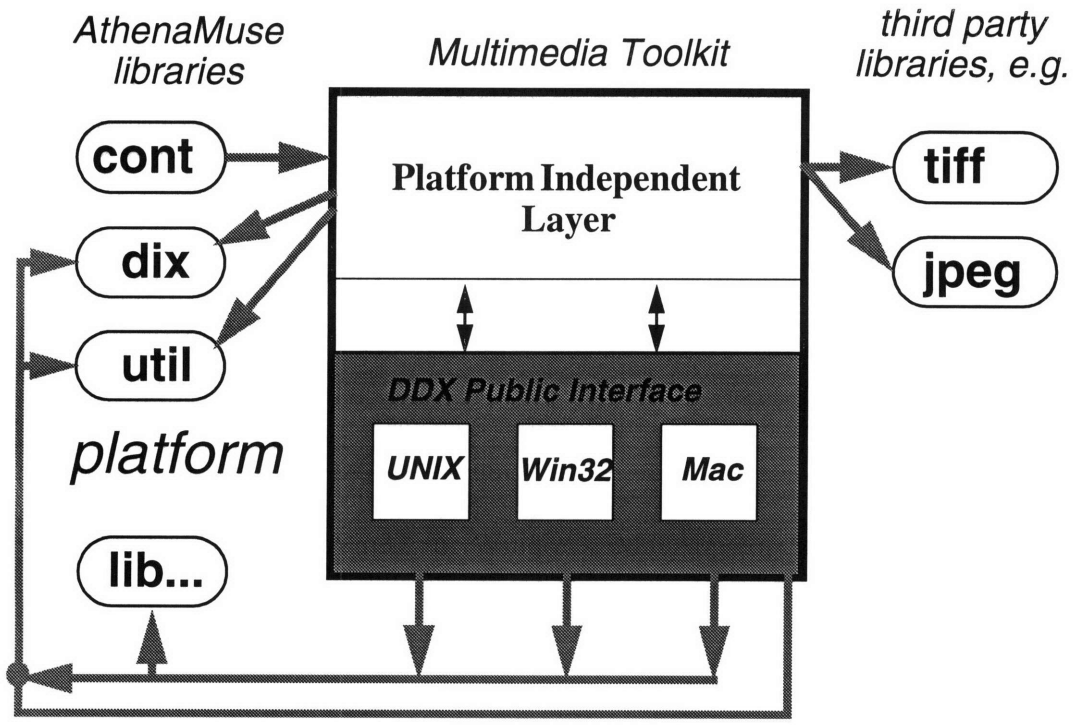

Figure 4.1: Multimedia Toolkit Library interface

This figure also shows the relation between MMTK and other libraries. Sometimes, certain services are needed for particular tasks. For example, one might need to be able to

read or write images in TIFF or JPEG formats. In this case, a third party library can be used to support these services.

Another set of libraries that MMTK uses are the DIX and the UTILS. These libraries are built in AM2 to supply basic services to other libraries such as data types manipulations in the UTILS library, and event and attribute handling mechanisms in the DIX.

4.2 AthenaMuse 2 User Interface

The user's perception of any application is based on interaction with the User Interface, making the user interface one of the most important components of an application. Many factors influence these perceptions. Does the application's user interface match the logical flow of the application? Are the interface components able to perform complex operations with simple interaction? Do the components behave as the user expects?

The author of an application requires the ability to easily define complex interactions between the various application interface components and the user. Providing the tools to support the authoring process requires a rich set of individual components, and the ability to combine these components to create reusable composite interface classes and modules. The author's view of the interface tools and components should be independent of the application's runtime window system.

The User Interface (UI) classes are the platform independent components that the Interface Control (XF) objects use to build application or module specific interfaces. AM2 uses the letters "*UI*" as a prefix for UI classes at the DIX level and the letters "*XF*" as a prefix for UI classes at the ADL level. For example a button at the DIX level is named *UIbutton*, while at the ADL level is named *XFbutton*. The UI objects do not have any direct understanding of or connection to control objects. The interface control object

which requests the creation of the UI object specifies communication with control objects through the registration of callback procedures.

The expected behavior of the various UI components is dependent upon the user's experience and the application's runtime environment. The different platforms that AthenaMuse will eventually be ported to each have a different "Look & Feel", which means that UI components that have the same logical use may look and perform differently on different platforms. Since AM2 is designed with the understanding that authored applications may be run on any of the supported platforms, the AM2 UI classes will be modeled on logical functionality. The individual platform dependent code will be implemented using the "Look & Feel" of the native windowing system. This model, while more difficult to implement than a single AthenaMuse "Look & Feel", will make AM2 more acceptable to users familiar with each of the supported platforms.

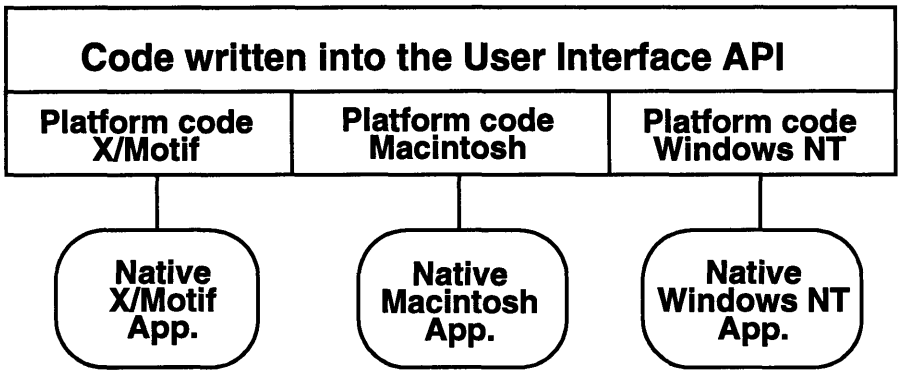

Figure 4.2: Portability Across Platforms

The UI module is designed to achieve the following goals:

- provides an object-oriented toolkit that lets the programmer build full-featured GUI applications quickly and easily.
- organizes the various components of an application (widgets, menus, fonts, etc.) into

a set of C++ classes.

- provides a platform independent interface to the control classes.
- provides platform independent abstractions of specific window system interface elements.
- combines platform independent code and private methods that are implemented using the "look and feel" of the native window system.

Following the general design of the MMTK, the UI has a layered structure (fig 4.2). The actual implementation is based on two sets of C++ classes: UI and UX (fig 4.3). The UI's are platform-independent and the UX's are the platform dependent classes.

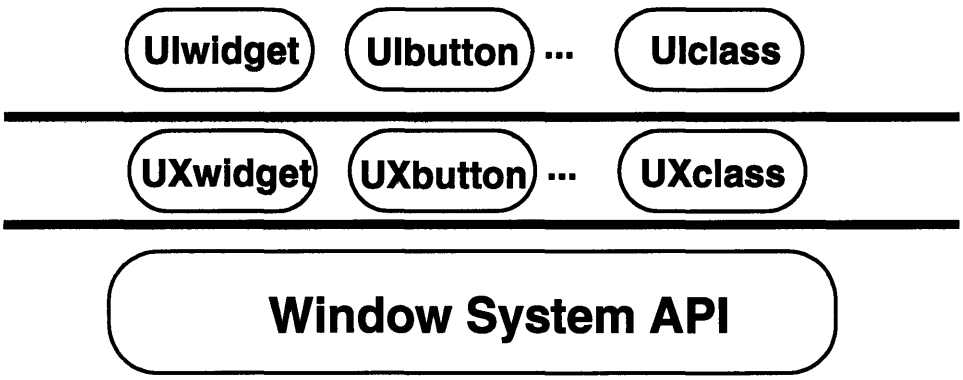

Figure 4.3: DIX &DDX classes

4.2.1 Class Hierarchy

There are four different categories of the UI classes:

- Widgets: such as windows, buttons and labels.
- Graphical classes: such as fonts and images.
- Special-purpose classes: such as menus and clipboards.
- Window system environment: for initializing the windowing system and media servers.

These categories are organized in three hierarchies depending on the functionality of the particular widget. These hierarchies are:

- Container Widgets: These are the widgets that can contain other simple types such as top level windows, sinks for images and sinks for hypertext.
- Simple Widgets: cannot contain any other widgets as their children. These are intended for some special purposes such as a push button or a label.
- Graphics and Special Purpose UI's: such as fonts, images and window system classes.

Figure 4.4 shows the full container widgets hierarchy. The classes that are visible to the user are:

- UItopShell: top level shell.
- UIvisual: a sink for different kinds of media.
- UIlayout: a layout manager for different widgets.
- UIhtml: a hypertext format for HTML documents.

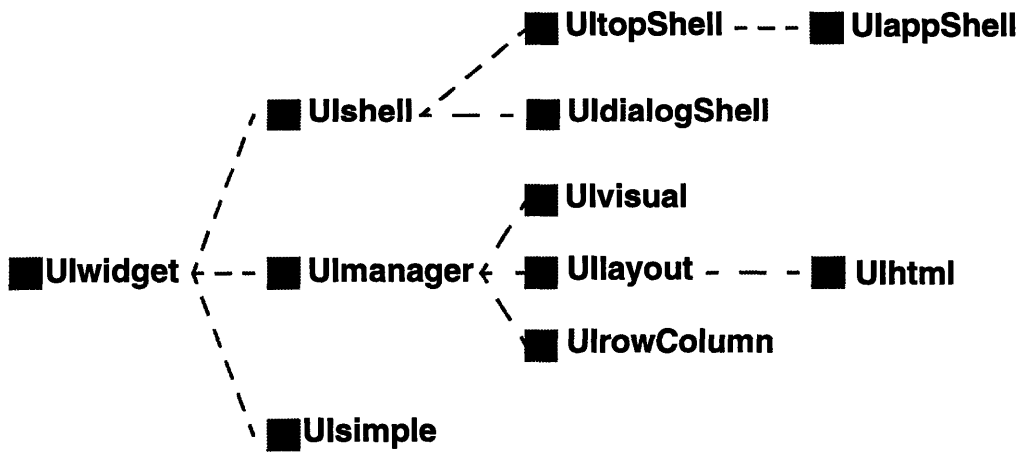

Figure 4.4: UI Container Widgets Class Hierarchy

Figure 4.5 shows the second hierarchy for simple widgets, the classes that are visible to the user are:

- UIlabel: a simple text label.
- UIbutton: a push button.
- UItext: a text widget.

For the third hierarchy (Figure 4.6) the classes that are visible to the user are:

- UIimage: images with more than two colors, could be 2 to 24 bits deep.
- UIbitmap: 1 bit images (only two colors).

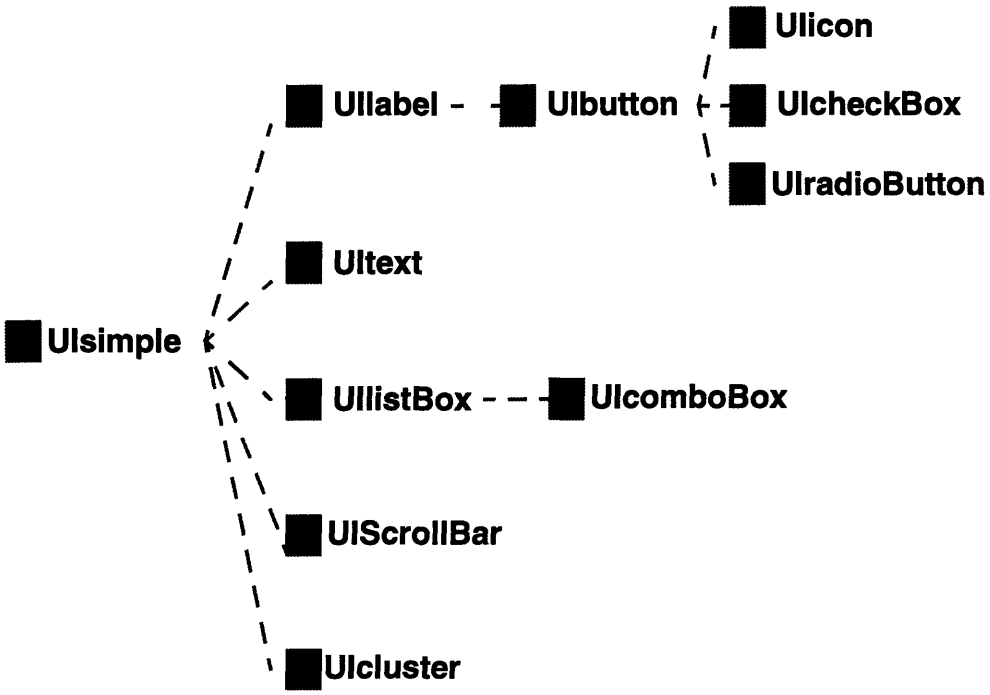

Figure 4.5: UI Simple Widgets Hierarchy

- UIfont: for font services. Normally one creates this font and attaches it to the appropriate text.
- UIwindowSys: responsible for initializing the windowing system and running the

main loop after the creation of the different widgets in an application.

The three hierarchies form a complete set of classes for building a GUI application whether they are used directly or from the ADL in which case the control engine of AM2 takes care of passing the routine calls between the ADL and UI objects.

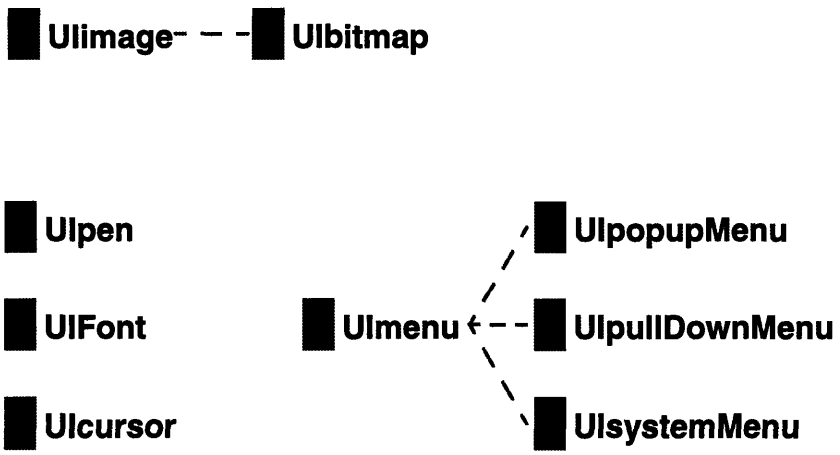

Figure 4.6: UI Special Purpose Class Hierarchy

4.2.2 The Attribute Mechanism

Setting and getting attributes of different UI elements, such as setting the width of a button or getting the background color of a test widget, is normally done in a platform-specific way. In order to cope with all these differences, AM2 makes use of a platform neutral mechanism for setting and getting attributes. The main goals of this mechanism are:

- avoid duplicate information: that is in order not to store the information about the attributes in more than one layer.
- dynamic use: an attribute can be added to a widget as needed. This will simplify the design a lot because for a particular window you may be interested in only one attribute or in twenty attributes (width, height, background, borderwidth,...)
- processing of multiple attributes at one time: this is important for UNIX because set-

ting one attribute at a time is very expensive.

- extensibility: if a new attribute is needed for a particular class, it can just be added without modifying the design.
- common interface between different modules: all modules have the same interface for setting and getting attributes.

The two classes implemented for this mechanism are the BSattributeMgr and BSattribute. In figure 4.7 below, the inheritance hierarchy is shown for the these classes. IN this diagram, an arrow from class C1 to class C2 means C1 *inherits from* C2, for example UIwidget inherits from BSattribMgr, and a line connecting two classes C1 and C2 with a circle on C1 side means C1 *uses* C2. For example, BSattribMgr uses BSattribute. Notice that all UI, Media and Network classes inherit from BSattributeMgr which in turn uses BSattribute.

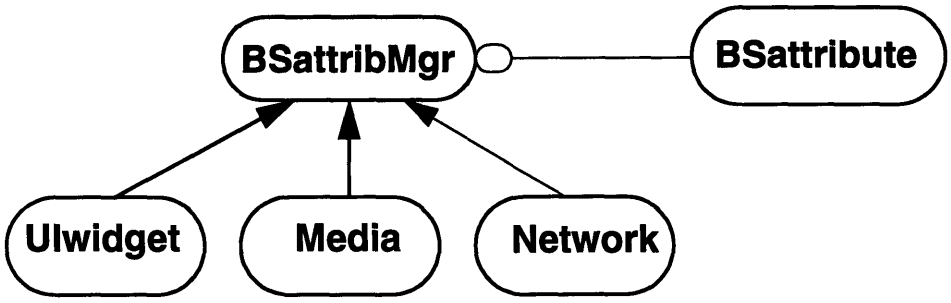

Figure 4.7: Attribute Mechanism Classes

Classes Description

Given below is a brief description of the classes for this mechanism.

BSattributeMgr:

This is an abstract class from which the UI widget classes (and some media classes) inherit. This class manages all the requests for setting and getting attributes received from

the control engine. The mechanism makes use of the following member functions of BSattributeMgr: (The types used are types defined in the UTILS library of AM2)

- *LookUpAttr*: returns the attribute's type.
- *InBatchList*: returns true if the attribute is in the list of attributes to be processed.
- *AddToBatchList*: adds the attribute to the list of attributes to be processed.
- *GetAttribute*: returns the value of the attribute retrieved from the windowing system.
- *SetAttribute*: sets the list of attributes queued in the batch list.

Notice that most of these methods are pure virtual since they should be defined in the derived classes.

BSattribute:

This class implements the attributes that are going to be set or retrieved. In other words, widget and media attributes are going to be visible to the control engine through instances of this class. Member functions involved in the mechanism:

- *GetValue:* returns the current value of this attribute. If the manager's batch flag is on, the returned value will be the one that it actually holds, otherwise it will be the one retrieved from the windowing system.
- *SetValue:* if the batch flag is off, it will ask the windowing system to set the value for this attribute immediately. If it's on, it will add this attribute to the batch list through the BSattributeMgr:AddToBatchList(..) function.
- *GetType:* returns the attribute's value type.
- *GetEntry:* returns a pointer to the attribute table entry that holds information about this attribute.

How to get an attribute

This is done in four steps. First, the client, who is the one requesting the attribute, calls the attribute manager with the GetAttribute method. Second, the attribute manager will create an instance of BSattribute. Third, this BSattribute will check to see if it is in the attributes batch list to get its value. If not it will retrieve the value from the system. Finally, this BSattribute will be returned to the client with the appropriate information.

The mechanism is illustrated in Figure 4.8.

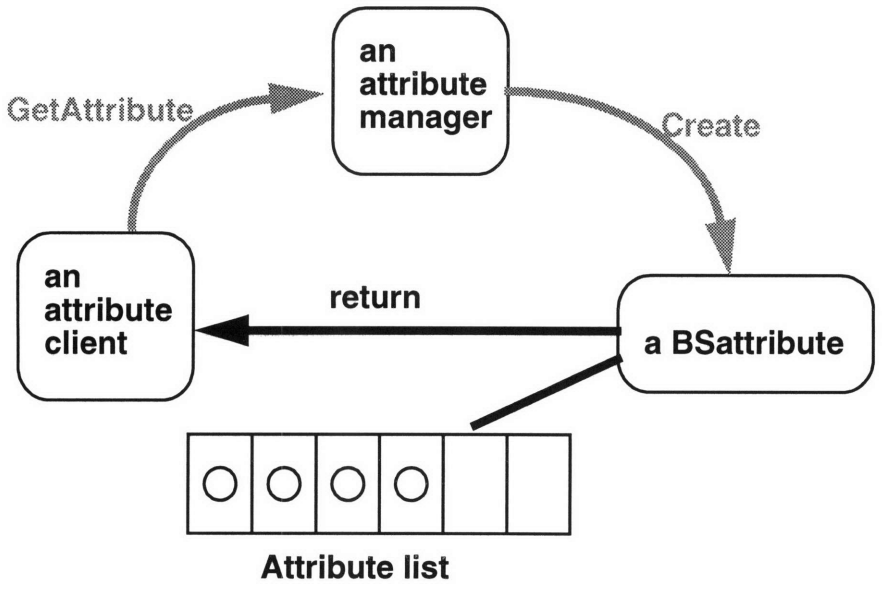

Figure 4.8: How To Get An Attribute

3.4.4 How to set an Attribute?

This is similar to getting an attribute. First, the client creates a BSattribute, then it calls the BSattributeMgr to set its value. This manager will add the item to a list of items to be processed called the batch list which is going to be flushed at the next logical moment. This means that, the attributes in this list will be updated later. The other list is the attribute managers list which keeps track of what attributes are attached to the particular UI component.

The figure below (Figure 4.9) illustrates this process.

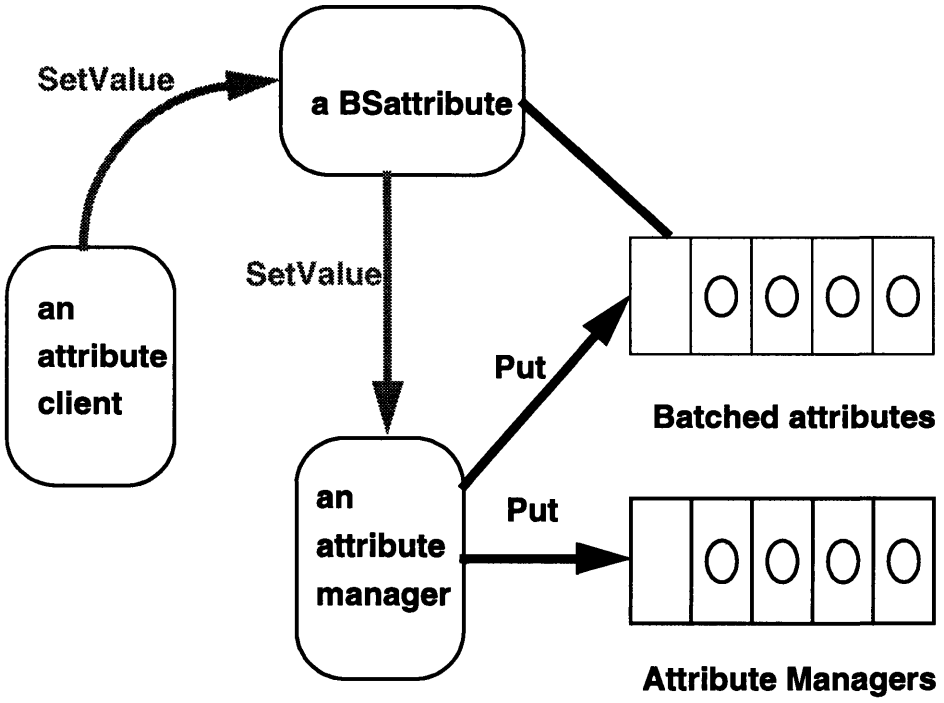

Figure 4.9: How To Set An Attribute

4.2.3 The Activity Mechanism

Similar to attribute processing, events are usually handled in a platform-dependent fashion. On Windows for example, there are messages and events. On UNIX, there are callbacks and on Macintosh, there are behaviors. Handling these interactions differs among these systems.

The AM2 activities manager is an agent that is responsible for registering for activities (or what is called in Windows terms an event or a message) with a certain widget in a way that if this widget receives that event, someone will know about that and do the required work.

The activities mechanism aims at three main goals:

- object-oriented approach: any event, message, or callback is modeled as a C++

object of a predefined activity class.

- extensible: adding new activities to the model should be easy and should not involve modification of the design.
- common interface between different modules: to register for an activity, the same interface should be used.

The classes implemented for this mechanism are the BSactivityMgr and BSntfnReq. In figure 4.10, the inheritance hierarchy is shown for these classes. Other classes are also designed in order to retrieve data from the activity. All these classes inherit from a base class UIactivityData. Notice that all UI's, Media and Network classes inherit from BSactivityMgr which in turn uses BSntfnReq and BSactivityData.

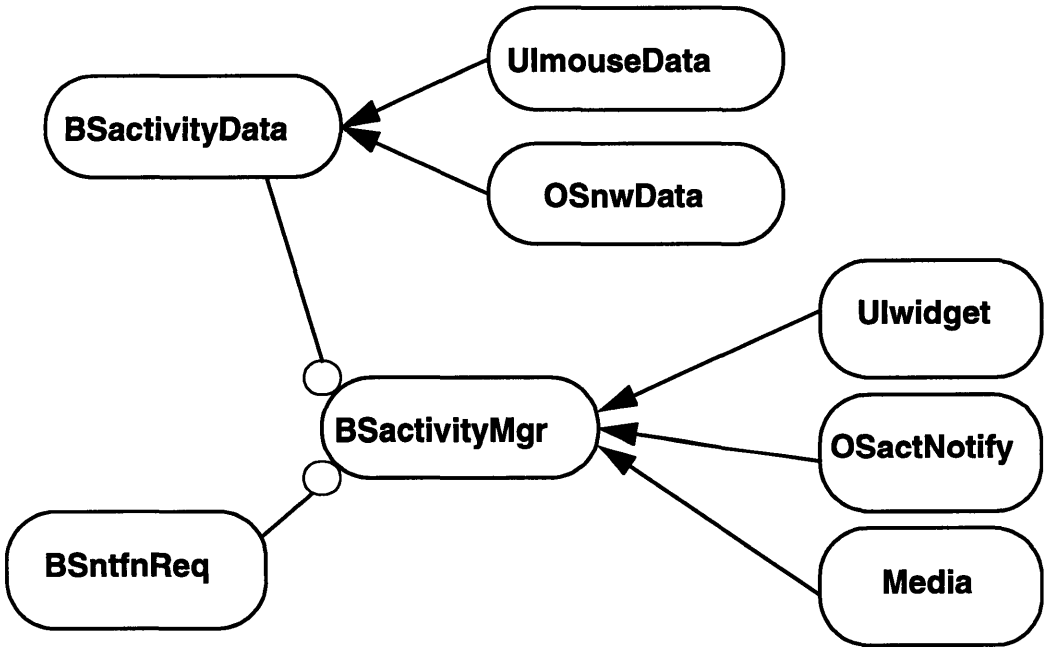

Figure 4.10: Activity Mechanism Classes

Classes Description

Given below is a brief description of the classes for this mechanism. Appendix A provides a more detailed description of the classes and their methods.

BSactivityMgr:

This is an abstract class intended to be the base class of any class that supports the subscription, unsubscription, and notification of activities. UI classes inherit from this class and therefore are able to support this kind of behavior.

In similar ways, some network and media classes are expected to be derived form this class as well. Basically, this class is responsible for managing the requests for notifications for the occurrences of activities (subscribe), the removal of such requests (unsubscribe) and the notification calls whenever the activities requested occur.

BSntfnRequest:

This is an abstract class that defines the common interface for the classes that will represent possible notification requests. Members used:

BSntfnRequestT:

This class implements a form of notification request. The notification request implemented in this class is one where there is an activity client object, an activity client's method, and some client data.

BSactivityData

This abstract class is the common interface for any representation of activity data.

UImouseData:

This class represents the data associated with different mouse activities. These activities include: MouseMove, MouseDown, MouseUp, MouseEnter, and MouseLeave.

UIrefreshData

This class represents the data associated with the refresh activity.

How Activities Work

The main mechanism can be seen as a two-step process:

First, an activity client subscribes (requests notification) with an activity manager on the occurrence of certain activity.

Second, the activity manager detects the occurrence of the activity and notifies the activity client.

In order to subscribe, the activity client needs to specify the name of the activity in which it is interested, what object is going to be notified when the activity happens, what method should be called in the notification process, and any client data to be passed to the client's method.

All this information is packed in an object called the notification request which is sent as an argument in the subscription process.

When the activity manager detects the occurrence of an activity for which there were any subscriptions, it notifies the client object(s) by calling the subscribed method(s). As part of the notification process, the activity manager sends any client data that was registered and any data associated to the activity that occurred.

The mechanism is illustrated in the figure below (fig 4.11).

An Example

The following is a short example of the activities mechanism usage in C++. Details from the window system initialization and widget's creation have been removed for clarity.

```
// Declaration of class ActivityClient which defines several handlers
// for different activities.

class ActivityClient {

public:
// Handler for activity generated when push-button is pressed
// void PushButtonHandler(BSactivityData * data, int *)
```

```
    {
      cout << "I was pressed!! " << endl;
    }


// Handler for mouse activities
 void MouseHandler(UImouseData * data, int *)
    {
      cout << "I got the message!! " << endl;
      cout << "Keys   : " << data->KeyList().Dump() << endl;
      cout << "values: " << data->ToValueList().Dump() << endl;
    }

// Handler for refresh activities
  void RefreshHandler(UIrefreshData * data, int *)
    {
      cout << "I got the message!! " << endl;
      cout << "Keys   : " << data->KeyList().Dump() << endl;
      cout << "values: " << data->ToValueList().Dump() << endl;
    }
};
```

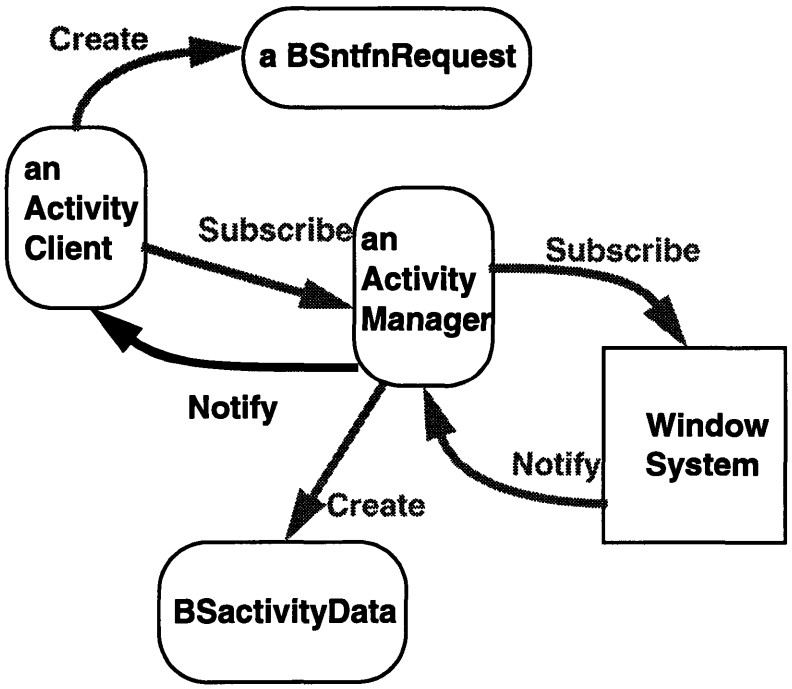

Figure 4.11: How Activities Work

```
main()
{
//-----------------------------------------------------------------
// Creation of an ActivityClient instance and several notification
// requests for different activities
//-----------------------------------------------------------------
  ActivityClient ac;
              int clientData = 123;

  BSntfnRequestT<ActivityClient, UImouseData, int>
    ntfn1("MouseMove", &ac, &ActivityClient::MouseHandler,
          &clientData);

  BSntfnRequestT<ActivityClient, BSactivityData, int>
    ntfn2("Pressed", &ac, &ActivityClient::PushButtonHandler, NULL);

//-----------------------------------------------------------------
// Creation of top shell and push button widgets
/-----------------------------------------------------------------

  UItopShell * top = new UItopShell( "top", 0 );
  UIbutton* btn1 = new UIbutton( "btn1", top );

//-----------------------------------------------------------------
// Subscriptions for notifications
//-----------------------------------------------------------------
  top->Subscribe(&ntfn1);
  btn1->Subscribe(&ntfn2);

//-----------------------------------------------------------------
// Unsubscribe requests for activities MouseMove and Pressed
// Only for demonstration!!
//-----------------------------------------------------------------
  if (!top->Unsubscribe(&ntfn))
    cout << " No subscription for that activity " << endl;
  else
    cout << "subscription removed " << endl;

  if (!top->Unsubscribe(&ntfn5))
    cout << " No subscription for that activity " << endl;
  else
    cout << "subscription removed " << endl;

}
```

The example shows how activities can be used in a totally platform independent way.

In the following section, we will compare this example to how Java does event handling.

4.2.4 Event Handling in Java: A comparison

Event handling in Java went through a major change from version 1.0 of the Java Development Kit (JDK) to version 1.1. The changes that were made brought the Java event handling mechanism closer to the AM2 activity mechanism.

In JDK 1.0, the model for event processing is based on inheritance. In order for a program to catch and process GUI events, it must subclass GUI components and override either the action() or handleEvent() methods. Returning "true" from one of these methods consumes the event so it is not processed further; otherwise the event is propagated sequentially up the GUI hierarchy until either it is consumed or the root of the hierarchy is reached. The result of this model is that programmers have essentially two choices for structuring their event-handling code[WWW3]:

1. Each individual component can be subclassed to specifically handle its target events. The result of this is a plethora of classes.

2. All events for an entire hierarchy (or subset thereof) can be handled by a particular container; the result is that the container's overridden action() or handleEvent() method must contain a complex conditional statement in order to process the events.

In JDK 1.1, a totally new model for handling events is introduced, named the "Delegation Event Model". The main reasons behind introducing this model were to overcome the problems with the old model, mainly the need for subclassing, merging all event types together and passing client data with events. The new model solves all of the these problems by introducing an approach similar to the activities approach in AM2.

Delegation Model Overview

"Event types are encapsulated in a class hierarchy rooted at java.util.EventObject. An event is propagated from a "Source" object to a "Listener" object by invoking a method on the listener and passing in the instance of the event subclass which defines the event type

generated.

A Listener is an object that implements a specific EventListener interface extended from the generic java.util.EventListener. An EventListener interface defines one or more methods which are to be invoked by the event source in response to each specific event type handled by the interface.

An Event Source is an object which originates or "fires" events. The source defines the set of events it emits by providing a set of set<EventType>Listener (for single-cast) and/or add<EventType>Listener (for mult-cast) methods which are used to register specific listeners for those events.

The event source is typically a GUI component and the listener is commonly an "adapter" object which implements the appropriate listener (or set of listeners) in order for an application to control the flow/handling of events. The listener object could also be another AWT component which implements one or more listener interfaces for the purpose of hooking GUI objects up to each other"[WWW3].

Similar to activities in AM2, a hierarchy of event classes is used to represent different event kinds. Each event class is defined by the data representing that event type or related group of event types. Since a single event class may be used to represent more than one event type (i.e. MouseEvent represents mouse up, mouse down, mouse drag, mouse move, etc.), some event classes may also contain an "id" (unique within that class) which maps to its specific event types. The event classes contain no public fields; the data in the event is completely encapsulated by proper get<Attr>()/set<Attr>() methods (where set<Attr>() only exists for attributes on an event that could be modified by a listener).

The delegation event model classifies events into two kinds: low level that deal with events at the windowing system level such as a mouse move or keyboard press; and

semantic which are defined at a higher level to encapsulate the semantics of a user interface component's model[WWW3].

Even though there might be some differences between AM2 activities and Java event delegation, they both seem to provide a clean and simple object oriented approach for event handling that gives a good logical separation between application and GUI code.

4.3 AthenaMuse 2 Media Engine

The AthenaMuse Media classes enable the author to create and use diverse media content without being concerned about the specifics of each data format or system specific access and presentation services.

AthenaMuse uses media element objects to describe the individual images, sound bites, film clips, text and other media units, which are included in an application. A media element is not the media technology but an object which represents the units of media that an author assembles into an application.

The term media data is used to represent the actual stream of information which is converted into the media presented (i.e., the frames from a video disc or a TIFF file). The media data does not reside within the media element. At run time, the media element objects retrieve the media data and perform the processing required to output the media.

4.3.1 Class Hierarchy

Media classes are divided into four categories:

- ME... Element: Individual media clips, sound bites, and other media chunks.
- MA... Access: Access to media data, and agents for access arbitration
- MD... Device: Device control
- MP... Presentation: Internal media data representation which interfaces with the presentation 'surface', or object

This categorization is very useful in separating what the media element represents versus its utilization. It also makes the media engine more platform independent. The reason for this is that many media elements may have the same presentation element, which is the part that communicates with the windowing system. Shown in Figures 4.12-14 the four hierarchies for the media module, the media access, the media presentation, the media elements.

The benefits of this classification are that it ensures more encapsulation and it provides commonly based classes which are interchangeable.

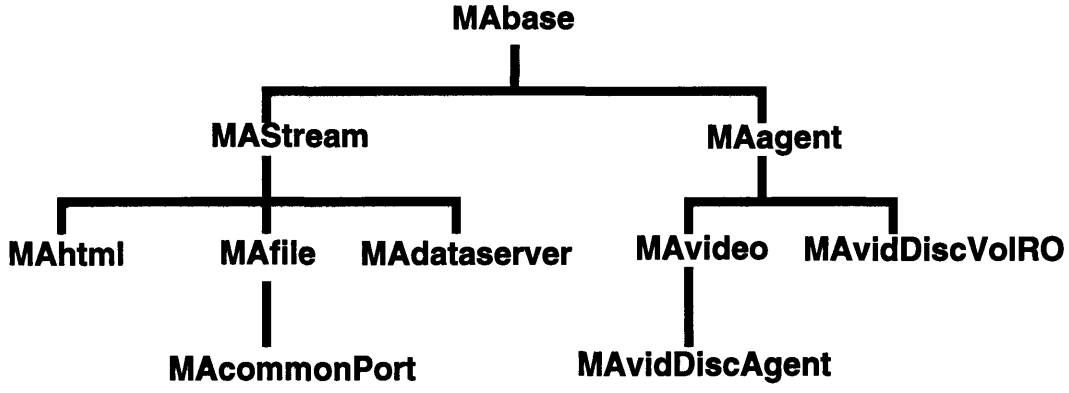

Figure 4.12: Media Access Hierarchy

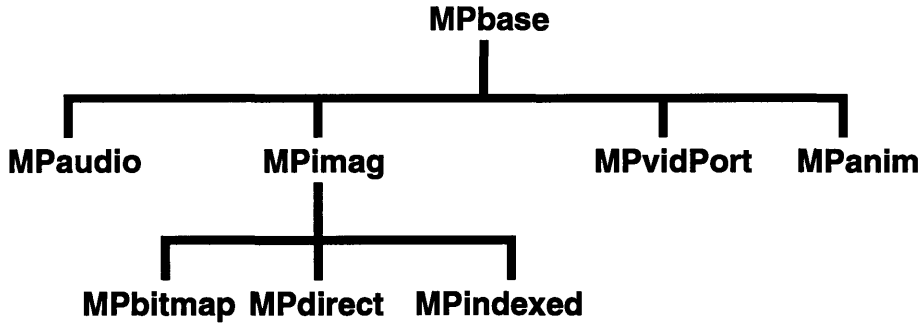

Figure 4.13: Media Presentation Hierarchy

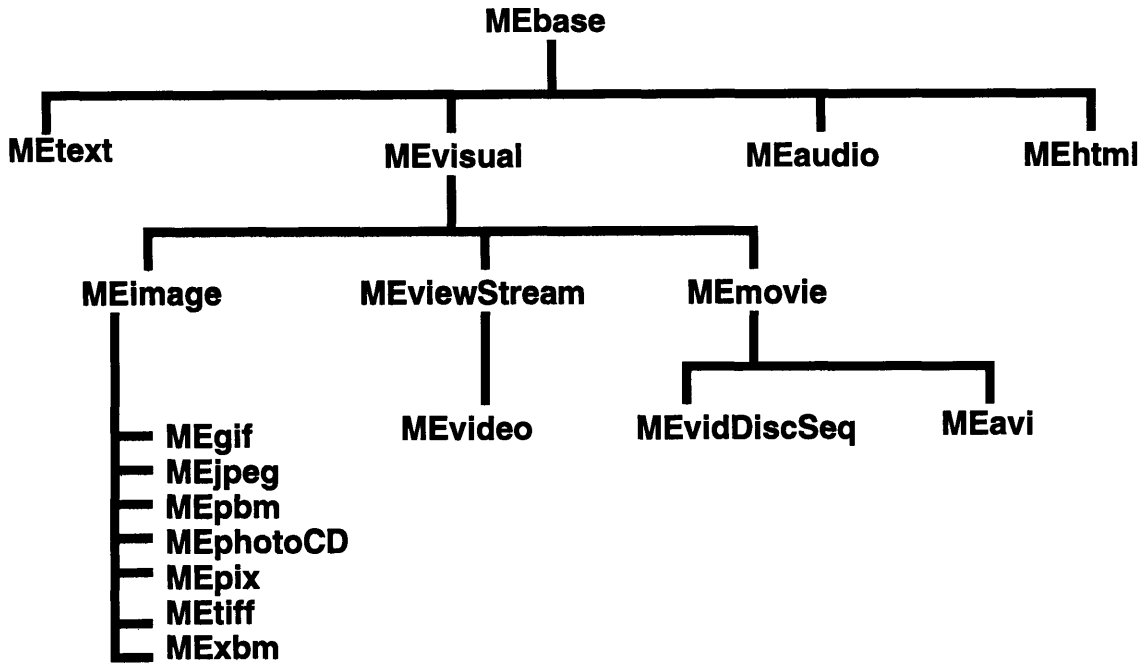

Figure 4.14: Media Element Hierarchy

Other classes that are not shown in the hierarchies are:

- MErgb -- single color, supports conversions between RGB, HSV, and internal 24/32 bit representations.
- MEnamedColor -- a named MErgb with a specific color value.
- MEcolorDB -- A collection of MEnamedColors, may be loaded from a file.
- MEbroker -- static class isolates ADL from details of media element construction.
- MEsink -- maintains presentation information for an element.

4.3.2 Temporal Media

In designing a multimedia library, temporal types of media need a very careful and clear design. This is important because of issues such as synchronization. The following design strategy is followed in the design of temporal media:

- Temporal media elements store position internally as absolute offsets from the beginning of the medium, using native frame units.
- Specification of an element's position, may use native frame units, normally as an offset from the beginning of the elements frames.
- Time position specification is independent of the native frame unit.
- Time positions are calculated using the default or element creation frame rate instead of the current frame rate.

Every temporal media element such as video or audio has the following attributes and methods:

Position:

All temporal media have a beginning, end, current position, and duration. Representation of these attributes should be consistent across the various ADL media types. Time based units provide the greatest level of interchange between different encodings of the same information, but time is subject to a number of factors.

The actual time required to present part of a media element may vary between different invocations of the presentation method, or by user modification of the playback rate. If time is to be the common representation of position within a media element, that representation of time should be derived from the optimal time that the position represents. Optimal time is defined here as the duration from the beginning of the media segment if the media were presented at its capture rate, or internal presentation rate. For example an NTSC videoDisc's internal frame rate is 30 frames per second, while an AVI digital video may have been created at 20 frames per second.

Many media elements may be defined using a sub-range of the data available, or of 'larger' media elements. The underlying media-specific class must keep track of these

absolute 'addresses' into the available data but the media element normally will use address relative to the segment when communicating with the ADL.Control a temporal media element is done through setting a list of attributes for the element or calling certain methods on it. The main attributes and methods are:

- *startPosition:* starting position of a temporal media element.
- *endPosition:* ending position of a temporal media element.
- *currentPosition:* current position of a temporal media element.
- *defaultRate:* at which the element is playing.
- *AudioLevel:* for elements with audio component.
- *Pause*: to stop the clip while keeping it on the sink.
- *Play:* to play from a certain positon to the other.
- *PlayFor:* for a period of time.
- *PlayUntil:* until you reach a certain frame number.
- *Step:* step to a certain position.
- *Notify:* notify someone at a certain time or by calling some activity.

This kind of control over temporal media elements allows for transparent merging of different kinds of media on different platforms. Having a media device capable of performing all the above requests on different platforms provides a good separation between the various kinds of media and the user access to them.

4.3.3 Mechanisms for the Media Engine

The same mechanism described for the UI are supported for the Media. As long as a class inherits from BSactivityMgr and BSattributeMgr, it has the ability to handle events and set attributes.

Something specific about media events is that they occur during the processing and presentation of media elements. Internal media events may indicate stages in the presentation of the media, activity in the data stream, and be derived from a media specific service, or generated by a timer.

4.4 Conclusion

In this chapter we covered different aspects of the MMTK in AM2 focusing on the overall design, the different classes used and the ability to provide a platform neutral interface to its users. Also, we gave a brief comparison between AM2 activity mechanism and the Java delegation event model.

In summary, the MMTK is an object-oriented platform-independent library that gives its users the ability to develop on one platform and directly port to the other platforms without having to rewrite the code for that other platform. In addition, this library hides from its users all the details of the windowing systems and thus makes it very easy to use. It has support for most of the media types required for a multimedia application and at the same time new media types can be easily integrated to it.

Chapter 5

Platform Independent Data Management

This chapter gives a general overview of several modules in AM2 that deal with data management, mainly data access and data transfer. From a user's point of view, data can be stored either in a file or in a database and can be moved across machines via some network protocol. Operating system services that support data storage and data transfer are usually specific to the platform, and in many cases the format itself might not be usable when the data is moved from one platform to the other. What is needed is a platform independent way for accessing and storing data.

In this chapter we highlight three aspects of AM2 that deal with platform independent data management: abstracting the file system in AM2, the database module, and then generalized data stream interface.

5.1 Abstracting the File System

In general, all platforms support the concept of storing data on disk in the form of a file which is visible both to the application and to the user. The access to a file, however, is quite different, and a platform independent interface is needed to access them.

From the platform-independent application's point of view, there must be a way to access a file on disk. When the file is opened and before it is closed, the system provides some reference to the file. From the user's point of view, a file is specified by a name and a location, which may be specified as a directory or folder.

Usually every platform offers a file system. In addition, C offers a library of file routines with wrapper functions around the operating system file functions. In Windows and

Macintosh, applications are encouraged to use the operating system functions directly and stay away from the C wrappers.

Specifying files must be done in a platform-independent manner. X and Windows use straightforward pathnames with a drive name on Windows, but on Macintosh, file referencing is a little more complicated.

From the Macintosh user's perspective there are desktop, volumes, folders, and files. Each has a name, which consists of 1-31 characters (1-27 for volumes that are disk partitions). The characters can be of any kind, except colon ':', which is reserved as a delimiter for applications that need to use pathnames or partial pathnames. Note that the slash '/' is a common character in file names, for example "Family Tragedies 12/24/92." The use of pathnames is allowed, but discouraged for several reasons:

- Since several mounted volumes may have the same name, a pathname may be ambiguous.
- Pathnames are unreliable as a means of identifying files or directories because the user can change the name of any element in the path at virtually any time.

A possible compromise in some situations is the use of partial pathnames. From a known directory, say the application's, we specify a path to the file we want. Although more robust than a full pathname, it would be useful if we were able to avoid it.

In AM2, a platform independent set of classes were developed that provide a clean access to files on different platforms. We will describe two classes here.

OSfileSpecification:

The class OSfileSpecification replaces the pathname on UNIX and Windows and the file naming on Macintosh. Its functions include:

- GetName: retrieves the file name.

- SetName: renames the file.
- GetPath: retrieves the file pathname.
- GetDrive: returns the drive number on which the file reside.
- Exists:checks if the file exists.

OSfile:

This class provides file input and output. It may be in either native or portable mode, written as text, binary, or tagged data.

Instantiating this class opens the file. All text written out is converted to a portable format (Unicode); binary data is not changed. It has several modes: ReadOnly, ReadWrite, WriteTrunc, and WriteAppend.

Some of its methods are:

- OpenNativeConstruct: opens the file. All text written out is in the machine's native encoding format.
- Close: Flushes all buffers and closes the file. Any further reads or writes fail until another file is opened with Open or OpenNative.

This interface for files allows for file access on different with the same code. AM2 input/output library takes care of passing requests for the file system with the appropriate machine specific routines.

5.2 The Database Module

One of the goals for AM2 authoring is the separation of application design and data. A further refinement of this goal is that it should be possible to reference interface and media objects symbolically in the authoring environment. The data needed to create the object should be stored externally. The AM2 database classes translate authoring language requests and persistent storage representations of AM2 application data into the data

required to create instances of AM2 objects.

The database classes (DT) are the means by which AM2 abstracts the different persistent storage models and methods. The database classes also manage the relationships between application data stored in an external database, such as a library inventory, and the data required within AM2 applications to present the inventory. Some of the main objectives for the database module in AM2 are:

- Separation of the database methods from the creation of AM2 objects.
- Database classes are built upon database independent methods. The goal of these classes is to provide a general interface which supports the persistent storage model via bindings to specific databases.
- Queries may automatically fetch all selected records, or provide a "cursor" to the Control Engine which may be used to fetch the specified number of records.

The database module provides a centralized and consistent mechanism by which data can be stored, accessed, modified and deleted. A database access language is used to specify data independent of the file system and other environment variables. Media objects can therefore be modified in the database, and these modifications will be reflected in the application, without needing to alter the actual code. Once all data belonging to a particular application is stored in a centralized location, it becomes much simpler to protect against accidental deletion and to move or copy the data as a single unit. Database systems are specifically designed for tasks such as data entry, deletion or modification. In addition, they also provide extensive facilities for data searching and querying that would not normally be available to an application. If selective data is required by an application it can be stored in the database and retrieved using a data query. By including a standard object-oriented interface, the application can be given access to preexisting databases, thus significantly increasing the usefulness and economic value of the application [CURT96].

In order to meet these requirements, AM2 makes use of a virtual database module (VDM). The virtual database module is designed to offer a standard method of interfacing with different database systems (figure 5.1) as opposed to the classical approach of building a separate interface for every database (figure 5.2).

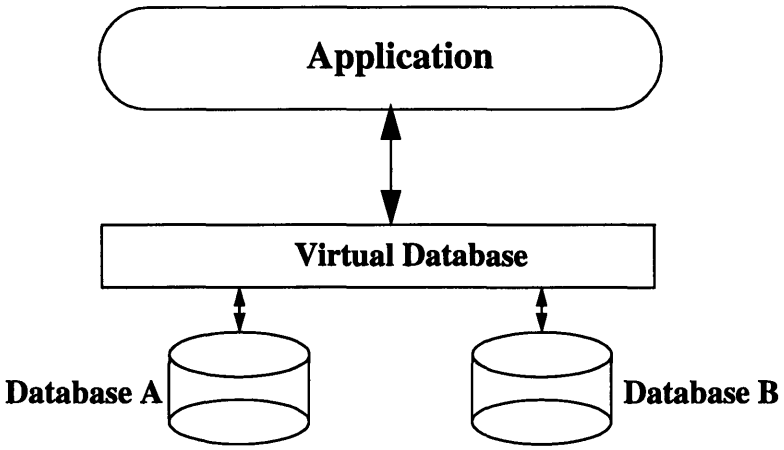

Figure 5.1: The virtual database p[CURT96].

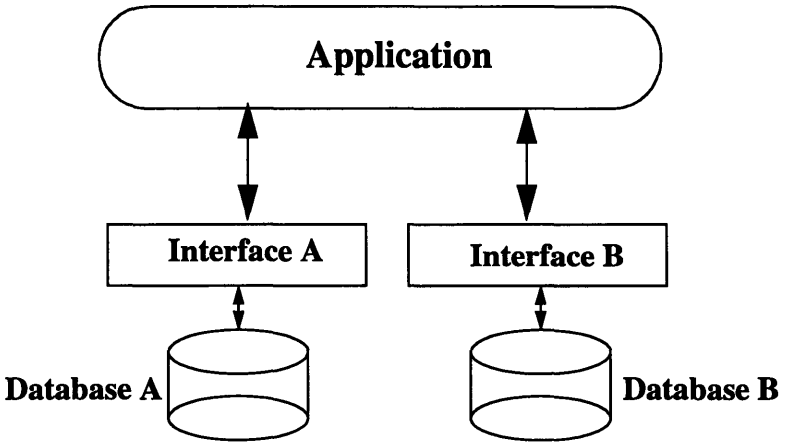

Figure 5.2: Using different APIs to connect to different databases[CURT96].

Using VDM, the structure of the database (schema) is retrieved automatically upon connection by the virtual database system. The application can then use this schema information to construct appropriate interfaces for the user.

A key component of any database system is the set of fundamental types from which class attribute and method definitions are built. An allowable data type in one of these definitions is either a fundamental data type or an existing class. The types defined for the VDM include simple types (e.g. integer, string), complex types (e.g. date, sequence) and multimedia objects (e.g. image, sound).

As to the interface, the VDM takes the form of a set of predefined database object classes. The designer can use these classes directly, or can derive more specialized classes for database manipulation.

The interface can be divided into two main parts. The first part is a set of classes that represent the fundamental data types available in the VDM, while the second part is a set of classes that represent a connection to a component database and provide operations for schema manipulation, query construction and query execution. The application author primarily uses an instance of the database class to establish a connection to a database and to perform various operations on that database.

A query class is available to aid in assembling and executing queries, while query results are accessed through a cursor object. A multidatabase class is also available to represent a set of database objects and allows the execution of simple multidatabase queries[CURT96].

Following the VDM approach for supporting databases does not only create platform independent interface for using databases but also a database-independent interface that allows applications to run using different database systems on different platforms.

5.3 Generalized Data Streams

The design and implementation of the network library in AM2 went through several changes during the life of the whole project. The need to access remote machines was first supported by providing a set of network classes that wrapped the TCP/IP protocol on the three platforms.

One key problem with this approach was that even though it provided network access, the interface was different from accessing data in a database or in a file. An application that needs to retrieve data from different sources has to have a separate implementation for each kind of data source.

What is needed is an interface that sits on top of different data sources and provides a general uniform way of accessing data for applications to use. A generalized data stream interface was proposed as a solution. This interface represents a general I/O model that is implemented as a layer on top of file I/O, network, database and Web services. The goal of the design is not only to allow streaming of data over the network but also to provide a unified interface for accessing information whatever storage it might be located on. A stream may originate from a memory buffer, a local file, an inter-process communication pipe, or a network connection.

The strategy behind the generalized stream interface addresses three problems. First, uniformity of interface to achieve independence of the media source; second, platform-independence based upon widely available services like FTP and HTTP and database access; and third, semantics for seeking into a stream.

The following three sections give some details of the stream model, the different classes implemented and goes in details on the design and implementation of the network stream that is one of the challenging elements of the generalized data stream interface.

5.3.1 The Stream Model

The Stream model incorporates the following:

- Stream Source
- Stream Destination
- Stream Unreliability

The stream source is modeled merely as a sequence of bytes at one end of the stream. This allows for different kinds of data sources to be valid, even multiple sources that are merged together.

The stream destination is modeled as a memory buffer. This does not impose any restriction on the destination itself; however, it adds some practicality in thinking about the model. It also allows us to define the stream behavior in terms of observations on the buffer. The memory buffer can represent any intermediate stage before the destination itself. The size of the buffer used by the stream can grow up to a maximum M defined as a characteristic of the stream. M specifies the maximum displacement in the destination buffer that a seek operation can successfully be done. For example, a video stream might set M to the size of a frame since no information about the previous frame is often needed when processing the current frame.

The stream unreliability lies in that seeking can not be performed beyond the bounds of the destination buffer. Of course, seeking becomes less important as the size of source becomes larger. More importantly, the stream unreliability is imposed by the fact that the stream will not guarantee that all the bytes at the source will ever reach the destination. For simplicity we assume that if at any time, a byte at the source is not delivered, every byte that follows will not be delivered too, i.e. the stream fails at one point to continue its operation.

A stream can receive commands from the user. These commands are the same as the commands used with a file descriptor namely open, close, read, and others. These commands are independently dealt with at the destination buffer level.

5.3.2 Class Hierarchy

In order to ensure uniformity of access for all kinds of streams, we differentiate between *streams* and *streamSpecs*. A *streamSpec* is a data structure that specifies the source of the stream. As an example, a *fileSpec* (subclassed from *streamSpec*) might contain the path to where a file resides, the name of the file, and some flags to indicate whether the file should be open for read or write or both. A *networkSpec* might contain a server name, a protocol to be used to connect to the server, a path and a file name.

Streams are constructed from their *streamSpecs*. All streams provide a uniform interface including methods such as *open, close*, and *read*. The only difference is how the stream will interpret these commands. A converter, that we call the *Stream Manager*, is responsible for converting a *streamSpec* into a real stream that provides this interface.

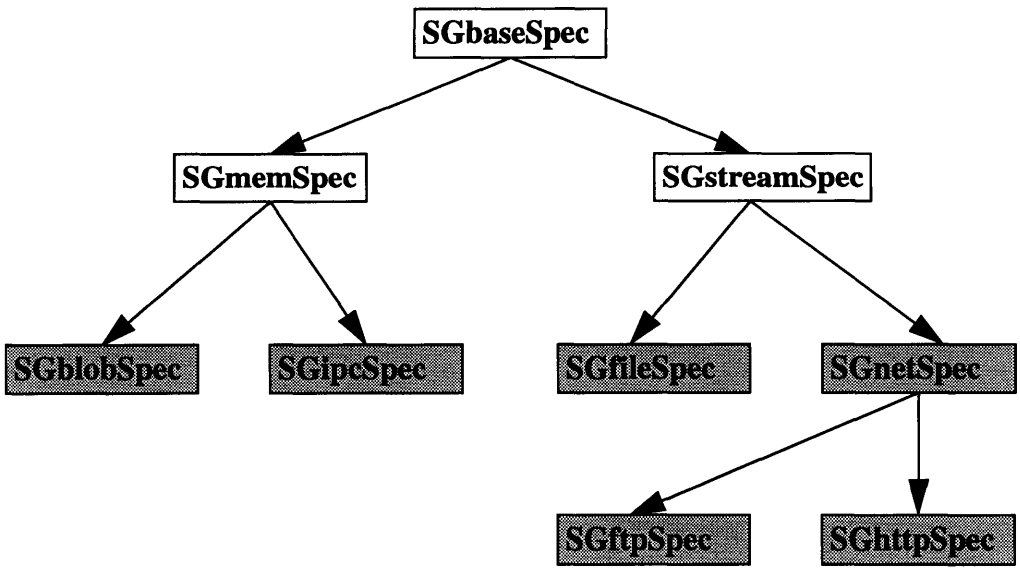

Figure 5.3: A hierarchy of StreamSpecs

Figures 5.3 and 5.4 illustrate the *streamSpecs* and streams hierarchies. The *SG* prefix stands for *Stream Generalized interface*. Dark boxes represent classes that can be instantiated whereas white boxes represent abstract classes.

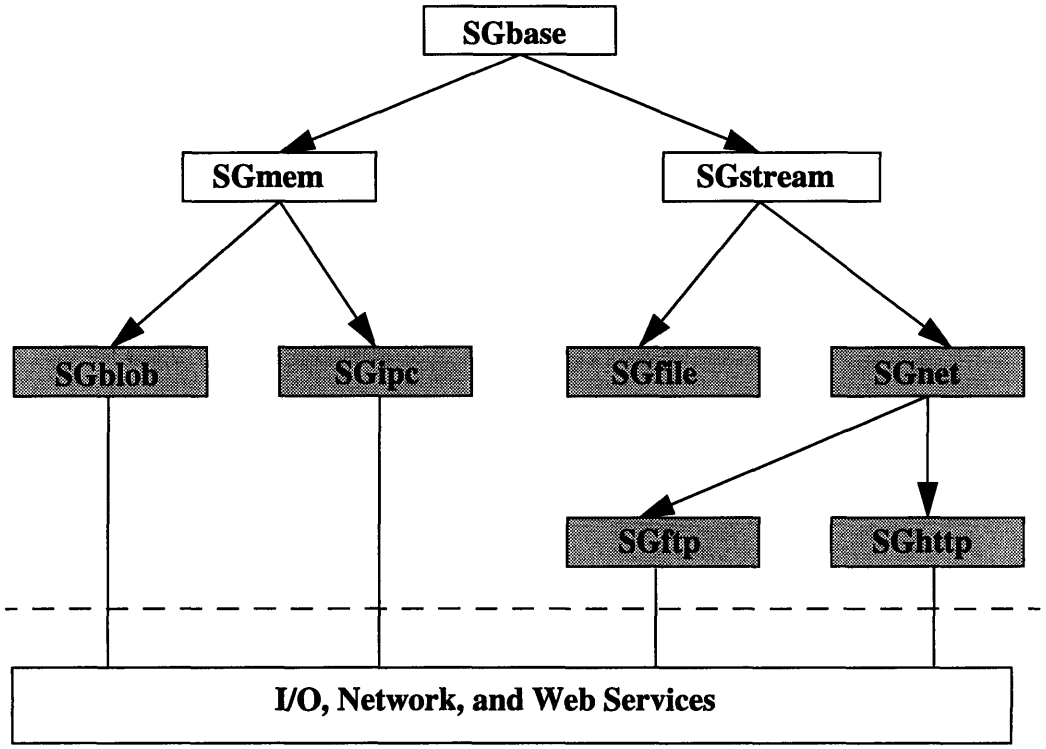

Figure 5.4: A Hierarchy of Streams

Different kinds of streams are derived from the base stream: blobs are simple memory buffers and IPCs are inter-process communication pipes. These two kinds of streams are Memory-Only streams in the sense that no disk access is required. Other streams contain files and network streams. The net stream is a general network stream, whereas ftp and http are more specialized versions of the network stream.

As it can be seen from the two figures, *StreamSpecs* form a parallel hierarchy to streams. The only difference is that StreamSpecs do not interact directly with services provided by the system. They only describe the stream source. The *Stream Manager* converts *streamSpec* into its functional stream. The Stream Manager needs to communicate only

with the interface of *baseSpecs* and base streams. Using an RTTI (Run-Time Type Identification) mechanism, the *Stream Manager* determines the actual level of the *Spec* in the *streamSpecs* hierarchy and creates the corresponding stream in the streams hierarchy. What follows is an example of constructing a file stream:

```
SGbase * file;
SGbaseSpec * spec = new SGfileSpec("fname");
file=SGmanager::StreamFromSpec(spec);
```

Any other kind of stream can be constructed and used in a similar way, thus providing the same interface and ensuring uniformity of data access across all streams.

The goal is to provide a cross-platform implementation of the Generalized Data Stream Interface. To achieve this, we use a platform-independent system services such as standard memory and file I/O operations, widely available network services such as FTP and HTTP, and third-party libraries available for different platforms such as the W3C library, which is a general purpose Web API written in C [FRYS95].

5.3.3 The Network Stream

The network stream is constructed by providing the *Stream Manager* with either an *SGnetSpec*, an *SGftpSpec*, or an *SGhttpSpec*. Both *SGftpSpec* and *SGhttpSpec* are specialized versions of *SGnetSpec* where the protocol to be used is defined a priori. The source of a network stream is a URL (Uniform Resource Locator). The URL is encoded in the *Spec* used for creating the stream. Since *SGftpSpec* and *SGhttpSpec* both inherit from *SGnetSpec,* the URL is actually contained in the latter.

The implementation of the network stream tokens is done using the W3C library available for UNIX and MS-Windows platforms[FRYS95]. The W3C library is a general-purpose Web API written in C that can be used as the code base for writing Web clients and

servers. The purpose of the library is to provide a sample implementation of HTTP and other Internet protocols.

5.4 Conclusion

AM2 makes use of a powerful set of modules to manage data independent of both the platform and the location of data. The use of a virtual database allows applications to access several kinds of databases simultaneously. Many applications that make use of the database module have been built and showed the effectiveness of the virtual database module as an efficient and easy way to built multimedia applications.

The Generalized Data Stream Interface based on the stream abstraction provides a convenient and uniform way of accessing data from different sources in a platform independent fashion. One good example is the network stream has been used successfully in streaming MPEG video.

Chapter 6

Lessons Learned and the Future

This thesis addressed the overall design and development process of cross platform software through a case study of the AthenaMuse 2 multimedia authoring environment.

Software developers building a cross platform application can benefit from following a coherent plan and a deep of understanding of the platforms the software will run on. The complexity of many platforms and their differences might make the development process extremely slow and inefficient in the absence of a formal design and development plan. Many lessons were learned from the design and development of AM2. We touched on most of those throughout the thesis.

Lessons Learned

The very first thing that should be well understood before developing a plan for cross platform implementation is what the software should do. Knowing the features required for the application will help in identifying the limitations of some of the platforms and where some serious work might be needed. AM2, being a multimedia authoring environment, required rich user interface and media support which motivated the idea of a separate toolkit for sophisticated user interface and media support.

Knowing what platforms offer is also another key issue. For instance knowing that two out of three platforms do not support some kind of user interface might lead to the decision of simply creating an emulated version of the user interface for the three platforms instead of trying to make two platforms behave like the third that supports the user interface. AM2 utilizes a set of user interface components that combine platform native features and some other emulated features, reflecting the fact that the common denominator

of the three platforms is not enough to implement the features needed for developing a multimedia application.

The availability of cross platform tools and libraries might also change the plans. For instance, having a cross platform network support from the W3C was a good choice for AM2 as a substitute for writing network modules for the three platforms. Because of the complexity of developing cross platform software, it is often better to look for specialized tools that offer solutions for one or more aspect of the software. AM2 makes use of several such tools for example, it relies on the Rogue Wave library for data structures which is available for the three platforms, and the TIFF image library for TIFF and JPEG that is also available for the three platforms.

Typically the largest part of the code that should be redone for platforms is in the user interface where no standards have been identified, and operating systems which provide different windowing systems with totally different programming and semantic interface. For AM2, most of the platform-dependent code is in the multimedia toolkit that has three distinct platform dependent (DDX) layers, one for each of the three platforms. Hence, special attention should be paid to designing a user interface that can be built across the targeted platforms with minimum effort.

Breaking the application down into a behavioral part and a data part is also important for a clean design. AM2 makes use of the virtual database module to manage the data part for applications where a large amount of data might be needed. The ability to store and retrieve data regardless of the kind of the database and from multiple databases at the same time brought AM2 a flexible and easy way of creating complex and multimedia-rich applications with a fairly small effort.

The ability to abstract data elements and manage them in a platform and source independent manner is also a key feature that the AM2 generalized streams provide. In today's

applications, data may reside locally, remotely, or in a database. Having to write separate modules or pieces of code for each is time consuming and not acceptable. The generalized streams in AM2 solve this problem in addition to providing a platform independent interface for applications to use.

The Future

So what does the future hold for cross platform software? Until very recently, the techniques given in this thesis were the most commonly used ones for developing platform independent software. In today's software world, almost every developer, if not using, is looking into or thinking of using Java. Among many other features, platform independence is one that Java offers. This is mostly what makes Java a very attractive option for developers. Using Java causes a paradigm shift from a machine-centric to a network-centric computing, thus making programs platform independent and only specific to the network protocol they are built on. Because of all the features that Java provides, the shift to using it looks very promising despite the fact that many parts of the language are still under development and the performance is still not acceptable for moderately large applications.

Appendix A

AM2 Activity Mechanism Class Description

A.1 BSactivityMgr:

This is an abstract class intended to be the base class of any class that supports the subscription, unsubscription, and notification of activities. The functions used are:

- ***Subscribe:*** subscribes the notification request pointed to by aNtfnRequest. Returns aNtfnRequest if the activity for which a notification was requested is valid, NULL otherwise. This method should be defined by the derived classes.
- ***Unsubscribe:*** unsubscribes the notification request aNtfnRequest and returns a pointer to it if it was successfully removed, NULL otherwise. This method should be defined by the derived classes.
- ***IsValidActivity:*** returns true if the activity referenced exists (is supported), false otherwise. This method should be defined by the derived classes.
- ***Notify:*** calls the appropriate member function in every object that requested notification for the activity referenced by this method. As part of the calling, it sends any data associated with the activity.

A.2 BSntfnRequest:

This is an abstract class that defines the common interface for the classes that will represent possible notification requests. Members used:

- ***DoIt:*** calls the notification method of the activity client sending the corresponding activity data.To be defined in derived classes.

A.3 BSntfnRequestT:

This class implements a form of notification request. The notification request implemented in this class is one where there is an activity client object, an activity client's method, and some client data. The signature of the activity client's method is determined by the type of the activity data and the type of the client data as in the following typedef:

```
typedef void (ACT::*ACMethod)(ADT*, CDT*);
```

where ACT is the type of the activity client object, ADT is the type of the activity data (e.g., UImouseData, UIrefreshData, etc.) and CDT is the type of the client data. Members used are:

- ***BSntfnRequestT:*** constructs a notification request object for the activity with name anActivity, activity client object pAC, method to be called aMethod, and client data pCD.
- ***DoIt:*** calls the notification method of the activity client sending the corresponding activity data pointed to by pData.This method returns int only to provide overriding facilities to be used in the timer mechanism.

A.4 BSactivityData:

This abstract class is the common interface for any representation of activity data. Members used:

- ***KeyList:*** returns a list object with strings describing the data keys.To be defined in

derived classes.

- ***ToValueList:*** returns a list containing the values of all the mouse activity data. To be defined in derived classes.

A.5 UImouseData:

This class represents the data associated with different mouse activities. These activities include: MouseMove, MouseDown, MouseUp, MouseEnter, and MouseLeave. The data for these activities is stored in:

- ***xmouse:*** x-coordinate relative to the widget's origin.
- ***ymouse:*** y-coordinate relative to the widget's origin.
- ***button:*** button pressed, if any.

Members used:

- ***SkeyList:*** returns a list object with UTstrings describing the UImouseData keys. Example: {"x", "y", "button"}.
- ***KeyList:*** calls the static SkeyList() member function described above.
- ***ToValueList:*** returns a list containing the values of all the mouse activity data.

A.6 UIrefreshData

This class represents the data associated with the refresh activity. The data for this activity is stored in:

- ***x:*** x -coordinate of the upper-left corner of the region that needs refresh relative to the widget's origin.
- ***y:*** y-coordinate of the upper-left corner of the region that needs refresh relative to the widget's origin.
- ***width:*** the width in pixels of the region that needs refresh.

• ***height:*** the height in pixels of the region that needs refresh.

Members used:

• ***KeyList:*** returns a list object with strings describing the UIrefreshData keys.

• ***SkeyList:*** calls the static Skmember function described above.

• ***ToValueList:*** returns a list containing the values of all the refresh activity data.

Bibliography

[NUTT92] Nutt, J. Gary, *Open Systems.* Prentice Hall, Englewood Cliffs, 1992.

[GLAD95] Glad, S. Anthony, *Cross-Platform Software Development.* Van Nostrand Reinhold, New York 1995.

[AM2D97] The MIT AthenaMuse Consortium, *AthenaMuse Release 2. Documentation.* Available from http://www-ceci.mit.edu, 1997.

[PETR94] Petrucci Steve, *Cross-Platform Power Tools,* Random House Inc., New Yourk, 1993.

[AM2D94] The MIT AthenaMuse Consortium, *Athen Muse 2 Design Specifications version 1.4.* Available from http://www-ceci.mit.edu, 1994.

[CURT96] Curtis K., "Multidatabase Support for Object-Oriented, Multimedia Authoring Environments," *Ph.D. Thesis*, Massachusetts Institute of Technology, 1996.

[FRYS95] Frystyk, H. N. *The W3C Reference Library.*, Available from the URL: http:// www.w3c.org/pub/WWW/library.

[MNEI97] Saadeddine Mneimneh, Issam Bazzi, Cyril Morcrette. "Generalized Data Stream Interface", *ATIRP First Annual Technical Conference*, January 1997.

[HRW94] Harward, V.J. and Lerman S.R., "The AthenaMuse Multimedia Environment," paper presented at the Massachusetts Telecommunications Conference, October, 1994.

[BRAI94] Brian, Marshall and Campbell Kelly, *Windows NT Programming: An Introduction Using C++.,* Prentice Hall, 1994

[WWW2] Campione, Mary and Walrath, Kathy., *The Java Tutorial.* Addison-Wesley 1997.

[WWW3] Sun Microsystems, *Java Development Kit 1.1.1 Documentation.* Available from http://www.javasoft.com

[ISAA94] Issak, James, et al, *Open Systems Handbook.* IEEE Standards Press, 1994

[EZZE93] Ezzel, Ben, *Windows NT 3.1 Programming,* Ziff Davis Press, Emerville,1993