Case Study on Selecting
An Environment for
Software Development

Justin Doak

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Los Alamos National Laboratory
Los Alamos, New Mexico 87545
Case Study on Selecting an Environment for Software Development

Justin Doak

1.0 Introduction

To explore the various issues and options surrounding software development, I have selected a specific Safeguards Systems Group (NIS-7) project to serve as a case study. The opinions expressed are solely those of the author and any reference to "we" or "our" refers to this single author. The goal of the selected project is to produce software that can accurately analyze data from sensors in tanks containing solutions of nuclear material (solution monitoring). This project focuses on data from Japanese reprocessing facilities. The software is to be used by International Atomic Energy Agency (IAEA) inspectors back at headquarters in Vienna after obtaining data from a site. I feel that the ideas presented in this paper may be applicable to numerous software developers whose project requirements are similar to those for this project as given in Section 2.0

Two considerations for developing software for use by others are discussed.

1. What software tools should be used during the development process?
2. What is the most effective way of distributing the software and documentation?

As an example of the first consideration, let's look at the choice of programming language. A good choice for a programming language is one for which there is a fully functional compiler on all the major platforms. To understand the type of issues we need to address under the second consideration, consider the choice of format for documentation. If we choose to distribute documentation in a postscript format, everyone who has access to a postscript printer can print a hardcopy of the document, so portability is not a concern. However, we may also wish that the documentation be viewed on-line and postscript viewers are not as readily available as postscript printers. So, we may instead choose to use Adobe's Portable Document Format (PDF). This choice also has its drawbacks as it requires that the reader have an Adobe Acrobat Reader. However, this is not a serious drawback as the reader can be obtained for free from http://www.adobe.com. In addition, postscript documents can be converted to PDF.

We briefly state our requirements for both of these considerations in the following section. Section 3.0 presents some options to meet the requirements and discusses the various trade-offs associated with each one. Finally, Section 4.0 gives our specific choices for the software development environment and means of distribution.
2.0 Requirements

We briefly state the requirements for our software environment and distribution of software and documentation in this section.

2.1 Environment

Those requirements relating to the software environment are listed below.

2.1.1 Portability

Portability means that the delivered software can be compiled, if necessary, and executed on the customer’s machine. Obscure programming languages, compilers, or other non-standard tools used in the development process make it difficult or impossible to use software on other systems.

Cross-Platform Compatibility. Our software must run on the three major platforms: Unix, Windows, and MacOS. In addition, we are extremely reluctant (usually unable) to maintain more than one development version (i.e., a separate version for each of the supported platforms).

2.1.2 Graphical User Interface (GUI) and Builder

A GUI front-end to our software adds considerably to its ease-of-use. This requirement also includes the use of a GUI builder that allows developers to easily create the layout of the interface. Once the layout is in place, the code that implements the interface can be automatically generated.

2.1.3 Creating Modular/Reusable Software Components

It must be easy to pull individual pieces out of a software system and use them for other applications. We have found that software that is not written with this requirement in mind often has so many interdependencies that taking out a section for use in another system is extremely difficult.

2.1.4 Generic Libraries

Libraries implementing generic functionality like lists, arrays, strings, etc. should be available to developers. This functionality may be provided with the language itself (e.g., the Standard C++ Library) or by third party software (e.g., Rogue Wave's Tools.h++[3]). Using a generic library should not decrease the portability of the software.

2.1.5 Environment Should Facilitate Development of Solutions to Large Real-World Problems

The choice of programming language most directly affects this requirement. If a language does not provide the capability to structure solutions to complex problems, it will not meet this requirement.

*Actually, the platforms that the software must run on depend on the customer. For example, one of our customers is the International Atomic Energy Agency where only Unix and Windows compatibility is required, not MacOS. In addition, our major software products, LANMAS and CoreMAS, only need to run under Windows.
2.2 Distribution
This section discusses those requirements concerning distribution of software and documentation.

2.2.1 No Special Privileges are Necessary to Access the Software and Documentation
Primarily, what we mean by this requirement is that the person does not need to have an account (i.e., user name and password) on our system in order to access our software.

2.2.2 Software Should be Visible to a Large Number of People
If no one knows that our software is available, it is not going to be used. What mediums will provide the greatest visibility for our software?

2.2.3 Documentation Should be in a Format that Everyone Can Read and Should Support Equations and Graphics
Our documentation is useless unless tools to read it are available on all major platforms. Furthermore, the original equations and graphics of the documentation must be supported by the format we choose.

2.2.4 Transmission of Software and Documentation Should be Hands-Off
What we mean by hands-off is that we, the developers, should not have to do anything to deliver the software and documentation once the means of distribution are set up. We would also like the ability to track who is using the software and documentation.

3.0 Discussion of Various Implementations to Meet the Requirements
We present candidate solutions to the requirements in this section. In addition, we address various issues pertaining to each solution.

3.1 Environment
Solutions pertaining to the development environment are covered in this section.

3.1.1 Portability
There are two components to portability: application and interface. The application software is written using a programming language. Obviously, some languages are more standard, and have better support, than others. For example, we would never choose to write our software in something like Icon, a little-known or -used programming language created at the University of Arizona. This is not a judgement on how useful the language is, but rather an acknowledgment that most customers for our software are not going to have compilers for this language. Standard languages include C, C++, Fortran, and Java, to name a few. Our choice for a programming language with which to develop our application software should be one of the standard languages with compiler support on most platforms. (With Java, it is required that the user have a run-time interpreter to execute software that is delivered in Java’s compiled architecture-independent byte-code format. See below.)
We provide here a quick note on the difference between compiled and interpreted languages. A compiled language requires that users have a compiler to translate the source code into an executable (i.e., machine code) that will run on their system. Of course, if the user has exactly the same environment (hardware and operating system) as the developer, then an executable can be delivered. Interpreted languages require an interpreter that understands how to run the source code on the user’s machine; there is no compilation phase. The user’s machine only understands machine code when an interpreter is available to translate the source code of the interpreted language.

**Is Java a Compiled or an Interpreted Language?:** Java is in-between object code and interpreted code. A Java class file is first compiled into the Java byte-code. However, the byte-code only runs on the Java virtual machine, so when a particular machine runs the byte-code, it needs a Java interpreter to make itself look like the Java virtual machine.

It is probably worthwhile to add a caveat or two to our previous discussion on the portability of source code for compiled languages. For the most part, if you write your application software according to the ANSI standards of a programming language, it should be relatively easy to port that software to another platform. For example, if we have developed software on a Sun using the SparcWorks C++ compiler and followed ANSI standards, the amount of code changes to get it to compile and execute on, say, a PC with a Borland C++ compiler should be minimal. Experience has shown that some changes will be necessary; however, they should not be too extensive. On the other hand, if we use functionality in the SparcWorks C++ compiler that is not ANSI standard, there is no guarantee that this same functionality will be supported under the Borland compiler. We may need to rewrite all sections of code that use this non-standard compiler functionality to successfully compile under the Borland compiler.

SunSoft takes a different approach to ensuring that third-party Java Development Kits (JDKs*) comply with the Java standard. SunSoft, for each new version of the Java language, will not give its Java stamp of approval to a JDK until it can demonstrate the required behavior on an extensive test suite. For example, if Microsoft wants to develop a JDK that runs under Windows NT, before Sun gives their stamp of approval, all the tools (e.g., the compiler, the interpreter) in the JDK must pass the tests in the test suite. This ensures that all JDKs provide exactly the same functionality as that defined by the Java creators, SunSoft.

Interface portability is a much more complicated issue. For example, it is impossible to successfully port software to a PC that was developed on a Unix system and contains direct calls to the Motif graphical library. The Motif library is not available on PCs. Another version of the software must be maintained that has the interface calls to the windows library on the PC. Obviously, maintaining a separate development version for each platform we wish to support is not good. In fact, one of our requirements is that we maintain only one development version. There are windows libraries that are cross-platform; the only catch is that the run-time interpreter for Java, as an example, must be installed on each system where our software is run to translate calls to Java’s Abstract Windowing Toolkit† (AWT) to native windowing calls.

---

*A JDK contains the entire suite of tools that make up the Java development environment. Some of the more frequently used tools in this package are the Java compiler, javac; the Java interpreter, java; and the applet viewer, appletviewer.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Another option is to write software as Java applets. Java applets are meant to be WorldWide-Web (WWW)-based applications that can be run from a Web browser such as Netscape Navigator. Java applets are first compiled into an architecture-independent byte-code format on the server. Then, when a client browser such as Navigator asks to run the applet, the entire byte-code is downloaded from the WWW server onto the local client machine and interpreted. This is the most portable of all development environments that we are aware of. The developer only needs to maintain one version of the software; this software is immediately available to everyone from the WWW server; and the only requirement to run the software is a Java-compatible Web browser.

However, there are some restrictions to Java applets. Namely, most Web browsers will not let the applet read or write files on the local (i.e., client) machine. If this is a requirement for the application, Java applets cannot be run through a web browser. Instead, an appletviewer that comes with a JDK can be used because it can be configured to allow read and write access to the local hard drive.

Netscape Navigator Note: Java applets, run from within Netscape Navigator and downloaded from a server, cannot currently read/write files on the local file system due to browser restrictions. Navigator 4.0 (or beta 2 of Communicator) may allow signed applets to read/write files on the local file system. In addition, even applets loaded locally from the user's CLASSPATH cannot currently read/write files; the reason is that Navigator loads files with the class loader, which enforces security restrictions, for both remote and local classes. The proper method of loading local classes is with the file loader that does not have the same stringent security restrictions as the class loader.

Cross-Platform Compatibility. Cross-platform issues are similar to the portability issues discussed above. Choosing a standard programming language for your application (e.g., C) virtually guarantees that a compiler will be available on any platform we wish to support. In an ideal world, the same code that compiles and executes on your PC will also compile and execute on your Sun, provided the code is written according to the standards of the language. The reality is that there are often numerous changes that need to be made to get the software to work on a different platform.

The interface issue is probably the most critical issue in determining whether or not our software will run cross-platform. Since we require only one development version, we can immediately rule out GUIs and GUI builders that make direct calls to the native graphical library on a platform. These systems require that you specify which platform you wish to support; then, they generate library calls to the native graphical library for that platform. The other option is to select a GUI that makes calls to a generic windows library, like Java's AWT, and lets the Java interpreter on the target platform generate the appropriate native library calls. The drawback here is that the target system must have the interpreter installed. Java has solved this problem by delivering a run-time interpreter for Java applets with software that is on virtually everyone's desktop, Netscape Navigator version 2.0 and above.

†The AWT is a set of generic GUI classes for writing window-based applications for a wide range of operating systems that support some form of windowing. For example, Windows, Motif, or the Macintosh.
3.1.2 GUI and GUI Builder

The functionality of our GUI builder takes a back seat to the need to have cross-platform compatibility (see above). Thus, it is essential to obtain a visual GUI builder that generates one set of library calls that can run on all platforms, provided the appropriate interpreter is installed. (Recall that the alternative is to generate code for each platform.) However, we do not want to require that the user obtain yet another piece of software to run our application. The ideal, then, would seem to be a cross-platform GUI library, for which a GUI builder is available, that is already installed on most platforms. The only cross-platform graphical library that is already available on most platforms is Java’s AWT, through the inclusion of the Java run-time interpreter in Navigator.

3.1.3 Creating Modular/Reusable Software Components

Given that many of our applications tend to be similar (i.e., data analysis), much of the software written for one application should be usable by another. Given current technology, we feel that the object-oriented paradigm and its ability to encapsulate functionality lends itself most readily to this requirement. There are numerous object-oriented languages available, including Smalltalk, C++, and Java, to name a few. We are currently using C++ primarily because of our past experience with C and the widely accepted use of C++. We are very aware, however, of the growing momentum for and acceptance of Java as the next generation object-oriented programming language.

3.1.4 Generic Libraries

There is currently great emphasis on quickly creating a version of the software that provides at least some of the required functionality (i.e., rapid prototyping). To this end, it does not behoove us to spend our time creating generic functionality like lists, arrays, and strings; we should obtain a well-tested library that provides this for us. All we really want is a description of the interface that we use in our programming. However, if we obtain a foundation library such as this from a third-party source, our customers must also obtain the library to use our software. The best solution appears to be that the language itself provides this functionality. The Standard C++ Library is an example of a programming language that has made generic utilities a part of the language itself. In this scenario it is not necessary to have both a compiler and the third-party library, but only the language compiler. The Java Application Programming Interface (API), the set of classes that come with the Java language, has a package of classes named java.util that will also provide this functionality.

3.1.5 Environment Should Facilitate Development of Solutions to Large, Real-World Problems

Another benefit of using an object-oriented programming language besides software reuse is that the paradigm is ideal for solving large and complex problems. Individual pieces of the problem can be isolated in classes, thereby avoiding the exploding complexity of large problems. We understand that object-oriented programming is the current state-of-the-art; however, we suspect that a new and even better paradigm will eventually appear. We are in constant surveillance of emerging technologies and will adopt them when appropriate.
3.2 Distribution

We discuss various options for distributing our software in this section.

3.2.1 No Special Privileges are Necessary to Access the Software and Documentation

This requirement narrows our choices for distribution considerably. Either we send users the software and documentation on some form of media or we provide an anonymous file transfer protocol (FTP) site. Requirement 2.2.4 above clearly states, however, that distribution must be hands-off; this means we cannot deliver the software through regular mail on CD-ROM or other media. Anonymous FTP appears to be the only remaining alternative. (Note that anonymous FTP can be accessed through a WWW browser by clicking on the file one wishes to transfer.)

3.2.2 Software Should be Visible to a Large Number of People

This is an easy decision; the WWW is clearly the most visible of all the mechanisms by which we can freely distribute our software. This requirement alone is enough to bias us toward using the WWW for all distribution.

3.2.3 Documentation Should be in a Format that Everyone Can Read and Should Support Equations and Graphics

What formats are readable on all the major platforms? Postscript, Adobe's PDF, and Hyper-Text Markup Language (HTML) can be read on most platforms. Although there are various versions of postscript, almost all versions can be printed to postscript printers. It is not the case, however, that all postscript documents can be viewed on-line; often postscript viewers are not available or do not recognize all versions of postscript. Adobe's PDF viewer, acroread, can be obtained free of charge from http://www.adobe.com and versions for all platforms are available. In addition, PDF format maintains the integrity of mathematical equations. HTML documents are readable on all platforms because of the prevalence of Netscape Navigator and other Web browsers; however, HTML does not currently support equations. A proposal for a concrete notation for HTML Math is almost complete with initial deployment via plug-ins.

3.2.4 Transmission of Software and Documentation Should be Hands Off

This requirement implies that transmission media like tapes, diskettes, and CD-ROMs are not an option; the developers have to actively package and mail the media to the end user. Almost the only way to make software and documentation available without having to actively distribute them is by anonymous FTP. One method of obtaining files when an FTP server has been set up is by using the FTP command to directly connect to the server and download the files desired. However, a more user-friendly solution is to create links from a Web page to the FTP files; the user need only click on the desired file and it is downloaded to his or her machine.

In 2.2.4, we mentioned that we would like to track who accesses our software and documentation. Most WWW servers create access log files that record various pieces of information (e.g., host name from which access originated). These can be used to calculate numerous statistics about accesses to a web site. In fact, there are numerous free utilities that process the access log files and calculate a variety of summary information.
4.0 Choices

Currently, we primarily write our applications in C++. In addition, we rely heavily on Rogue Wave’s Tools.h++ library for generic utility classes like strings, lists, arrays, etc. Some factions within our group use XDesigner as their GUI builder; it allows the visual creation of the user interface and generates C++ classes. A significant portion of the group has postponed the decision on a GUI and builder because of rapid changes in this area in the software industry. We have just recently set up an anonymous FTP server that, in conjunction with our WWW server, will allow users to download our software and documentation with a Web browser. There is no consistent format used for our documentation; if there is any, it is usually in ascii format.

We have a radically different vision for the future of software development and distribution given new tools and the growing demand for WWW-based applications. We will structure our vision according to the requirements in Section 2.0 on page 2 and the Issues in Section 3.0 on page 3.

4.1 Programming Language

We feel that C++ is a fully functional object-oriented programming language and is a reasonable choice. Java also appears to have all the capabilities we need in an object-oriented language while avoiding some of the pitfalls of C++; in addition, Java allows you to create WWW-based applets, increasing the portability of your software.

4.1.1 Requirements Met

2.1.1. Java and C++ are both standard programming languages and adequate compilers (and run-time interpreters in Java’s case) are available on all the major platforms.

2.1.3. Both Java and C++ are object-oriented programming languages that support the notion of a class. Classes are a mechanism that strongly encourages software reuse.

2.1.4. C++ has the Standard C++ Library (specifically, the Standard Template Library) and Java has its API (specifically, the java.util package). Both provide generic functionality such as lists and strings.

2.1.5. The object-oriented paradigm supported by Java and C++ enables structured solutions to complex real-world problems.

4.1.2 Our Selection

We select Java as our programming language of choice, primarily because of the increased portability afforded by the notion of an “applet,” but also because we feel the language has made significant improvements over C++.

4.2 GUI and GUI Builders

Our initial decision was to obtain a commercial GUI builder that would generate C++ code and be cross-platform. However, this approach requires that we maintain a different version of the software for each platform we wish to support, so it was discarded.
Another option is to write the interface in Java, but the underlying application remains in C++ at the server. Remote procedure calls are executed from the client to access functionality at the server. Provided that the target systems, regardless of platform, have a Java-compatible browser, they can run the application. In addition, we are still maintaining only one development version.

The best long-term option may be to write the entire system, application and interface, as a Java applet and make it accessible on a WWW server. Although this provides the most flexibility (i.e., only one development version that does not even need to be shipped to customers), it may take several minutes for clients to download the Java architecture-independent byte-code for large applets. This performance hit only occurs when the applet is downloaded the first time; after that, a cached version of the applet is used. Another potential problem is that many applications will need read and write access to files on the client machine; this is not currently allowed by most Web browsers. The temporary solution to this problem is to install a JDK, freely available from http://java.sun.com, which contains an appletviewer, on the local machine. Security restrictions for applets run with the appletviewer are much less strict than for applets run with a browser, and reading and writing of files on the local system is allowed, provided certain preparations are made. For more information, see http://www.javasoft.com/sfaq/#allowRead. A more long-term solution is to sign applets so that they are trusted by the client browser thereby allowing more intrusive access to the local file system. Netscape Navigator 4.0 is supposed to support signed applets.

4.2.1 Requirements Met

2.1.1. Java applets are by far the most portable option since the only requirement is that someone have a web browser containing a Java run-time interpreter.

2.1.2. There are several GUI builders that generate C++ code (e.g., XVT and XDesigner), but they are not acceptable because of the reasons mentioned above. Visual builders for Java are becoming more and more prevalent. One recent entry into the market is JFactory by Rogue Wave.

4.2.2 Our Selection
Java's portability again motivates us to select it as our GUI. Although GUI builders for Java are still primitive, they are expected to improve rapidly; Symantec and Rogue Wave have satisfactory builders out now for PC platforms.

4.3 Libraries Providing Generic Functionality
We currently rely on Rogue Wave's Tools.h++ for generic functionality like strings, lists, and arrays. Unfortunately, this means our customers must have the Rogue Wave libraries installed on their system or they cannot use our software! Although Rogue Wave claims that all C++ compilers will eventually come with Tools.h++, we feel that this is not likely. The Standard C++ Library, which includes much of the same functionality as Tools.h++ in the Standard Template Library, will come standard with all compilers since it is part of the language. The same can be said for the java.util package in the Java API.

*There are other JDKs available besides the one provided by Sun.
4.3.1 Requirements Met

2.1.1. The Standard C++ Library is extremely portable since it is actually a part of the language; if you have a C++ compiler that supports ANSI standard C++, you have the Standard C++ Library. Java also has a generic library built-in to the language. Tools.h++, on the other hand, is not nearly as portable since it forces our customers to first obtain the Rogue Wave libraries.

2.1.4. Obviously, the generic library requirement is met by all of the tools mentioned above.

4.3.2 Our Selection

The Standard C++ Library and the Java API are both very powerful and portable ways to meet this requirement. However, given the other benefits of using Java, we will use the java.util package to provide our generic library.

4.4 Distributing Software and Documentation

Anonymous FTP seems to be the method of choice for distributing software and documentation. In addition, when this mechanism is tied into the WWW, a very powerful distribution system is created combining the visibility of the Web and the ease of anonymous FTP.

4.4.1 Requirements Met

2.2.1. Anonymous FTP does not require any special privileges, like a password, for access.

2.2.2. The WWW may be the most widely viewed medium on the planet at the moment; second only, perhaps, to television. However, it is not possible (yet) to download software from your TV.

2.2.4. Anonymous FTP is as hands-off as it gets. We, the developers, do not have to do anything to distribute our software once things are set up.

4.4.2 Our Selection

Set up an anonymous FTP server and a WWW server. Provide access to all software and documentation through an appropriate Web page.

4.5 Documentation Format

Given our desire to publish software and documentation on the Web, a natural decision is to use HTML format for our documents. However, at this point in time, HTML does not have adequate equation support. We feel that Adobe PDF or postscript formats are reasonable alternatives in the meantime since both formats can be downloaded and printed or viewed on a client machine. (As mentioned previously, a proposal for a concrete notation for HTML Math is in the works.)
4.5.1 Requirements Met

2.2.3. Both PDF and postscript support equations and graphics and can be read on all platforms.

4.5.2 Our Selection

Use PDF or postscript format for distributing documentation at the current time. When HTML provides adequate equation support, we will migrate to it.

5.0 Summary

We can summarize by saying that there are several options competing to become our next development environment. One option is to continue to write our application software in C++. We have considerable experience with this language and have found it to be an excellent problem-solving tool. Another advantage is obviously the Standard C++ Library. On the other hand, the user interface issues remain. We cannot use a GUI builder that generates C++ classes as this requires that we maintain a different version of our software for each platform.

Another option is to write the interface in Java, but the underlying application remains in C++ at the server. Remote procedure calls are executed from the client to access functionality at the server. Provided that the target systems, regardless of platform, have a Java-compatible browser, they can run the application. In addition, we are still maintaining only one development version.

Our final option is to write everything, application and interface, in Java. Java appears to be as powerful a programming language as C++ and the concept of a Web-based applet is extremely attractive from a portability perspective. (To run a Web-based applet, the only requirement is a Java-compatible Web browser.) There are potential problems with Java applets. One is that an applet run from a Web browser cannot currently read or write files on the machine that is executing it. This problem can be circumvented by installing the JDK directly on the user's local file system for access to the less restrictive appletviewer. Also, future browsers are expected to allow the signing of applets and will give these trusted applets more access to the local file system. Applets, since they are interpreted, will run slower than a compiled version of the same software. As the technology improves, the gap between interpreted and compiled code in terms of speed is narrowing. In fact, Just-In-Time (JIT) compilers are currently available and provide a significant speed-up for Java code. We feel that what we gain in portability far outweighs any performance loss we might see.
6.0 References

We only list a few of our references here as many pointers are given to Web sites throughout the document.


Appendix A Step-by-Step Procedure for Software Development and Distribution

These are the steps the developers must follow.

1. Create the Java classes that implement the interface and functionality for the applet.
2. Compile the classes into Java’s architecture-independent byte-code format.
3. Embed the applet on a Web page at the developer’s site and send the user the Uniform Resource Locator (URL). If the application does not require the reading or writing of files on the machine that is executing the applet, go to Step 6. Otherwise, go to Step 4.
5. Instruct the user to modify his properties file to allow applets to read and write the appropriate files or directories on his machine. See http://www.javasoft.com/sfaq/#allowRead for more information on the necessary modifications to the properties file.
6. Put documentation, if equations are not necessary, as HTML files on the WWW server. If equations are embedded in the documentation, put the documentation in postscript or PDF format and place it on the Web server at the development site available via anonymous FTP.

These are the steps the end user must perform if the applet needs to read or write files on his local machine. Otherwise, the end user simply needs to open his Java-enabled browser and go to the WWW page that contains the applet; the applet will automatically start running once the Web page in which it resides is opened.

1. Install the JDK for access to the appletviewer.
2. Modify the properties file.
3. Open the URL in which the applet is embedded with the appletviewer.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.