A Java-Based DVI File Reader

Ce Guan

A Thesis
in the
Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements For the
Degree of Master of Computer Science

Concordia University
Montreal, Quebec, Canada

June 2005

©Ce Guan, 2005

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-10285-3
Our file Notre référence
ISBN: 0-494-10285-3

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

ABSTRACT

A Java-Based DVI File Reader

Ce Guan

PowerPoint is a convenient tool for people who work with other Microsoft software on Windows platforms; it is less useful to those who work with (La)TeX on UNIX systems. University users who tend to produce high quality slides by using (La)TeX prefer files to be displayed by a computer rather than printed on foils. Therefore, the idea behind this practical thesis is to understand the format of DVI files, since more and more DVI files currently available from many ftp sites, and build an application that provides highly stable service, functioning like PowerPoint, to view high quality slides of DVI files on Windows platform.

The *DVIReader* is built on an architecture that separates functionality from concrete document format, is a viewer for (La)TeX 's generic output format known as .dvi, and presents the "pages" of the .dvi file one at a time as a slide show on the Windows platform. Also this thesis presents an object-oriented design methodology that is used to implement this architecture using JAVA application technology, and an implemented tool that supports all of the above functionality and more.

ACKNOWLEDGEMENTS

I have prepared this thesis under the supervision of Prof. Peter Grogono. I am truly indebted to him for his constant encouragement and valuable guidance without which I would not have been able to complete my research successfully.

Last but not the least; I thank all my friends and family who supported me in this endeavor of mine.

Chapter 1 Introduction 1

1.1 Basic concept of DVI format 2

1.2 Object Orientation and Java 5

1.3 Overview and Requirements of DVIReader 6
1.3.1 Working process 6
1.3.2 Standard File Formats Required 7
1.3.3 Flexibility 8

1.4 Related Work 8

1.5 Organization 13

Chapter 2 LaTeX and the DVI File Format 14

2.1 Getting Start with LaTeX 14
2.1.1 TeX and LaTeX 14
2.1.2 A Typical LaTeX Input File 15
2.1.3 Characters and Control Sequences 17

2.2 Documentation for the DVI file format 18
2.2.1 The DVI File Format 18
2.2.2 DVI Format 21
2.2.3 Table of Opcodes 22

2.3 User Interface Design 24

2.4 Module Description 25

Chapter 3 Objected-Oriented Design and Implementation of DVIReader 26

3.1 System Architecture 27

3.2 In-Depth View of Design 29

3.3 Detailed Design Description 32
3.3.1 User Interface Design 32
3.3.2 Major Class Diagram and Key algorithms 36

3.3.3 Sequence Diagram in System..47

Chapter 4 Testing Plan and Results.......................49

4.1 Environmental Testing...49

4.2 Functional Testing..50

Chapter 5 Conclusion and Future Work..............52

5.1 Contributions..52

5.2 Future Research Directions...53

Bibliography...55

Appendix a Description of Opcodes [3]................57

Appendix B User Manual and Installation...........65

List of Tables and Figures

Figure 1.1 cmr12 scaled 7000 which looks in DVI---------------------------------4
Figure 2.1 Text View in LaTex---16
Figure 3.1 Architecture Design Diagram---28
Figure 3.2 System Class Diagram--29
Figure 3.3 Main Window--33
Figure 3.4 Result Window 1---34
Figure 3.5 Result Window 2---35
Figure 3.6 MainWindow Class---36
Figure 3.7 *DVIReaderContext* Interface Class---------------------------------------37
Figure 3.8 DVIReaderPanel Class--38
Figure 3.9 *DVIReaderDocument* Class---39
Figure 3.10 *DVIReaderRender* Class---40
Figure 3.11 *Bitmap* Class---41
Figure 3.12 *Font* Class---42
Figure 3.13 *Glyph* Class--43
Figure 3.14 *NybbleInputStream* Class--44
Figure 3.15 Sequence Diagram for System Activities-----------------------------47
Figure 3.16 Sequence Diagram of Font Module------------------------------------48
Table 2.1 Tables of Opcodes---24
Table 4.1: Test Cases of *DVIReader*--51
Table A: Shortcuts Descriptions of *DVIReader*------------------------------------65

Chapter 1 Introduction

In this chapter, we introduce basic concepts and terminologies with regards to our tool *DVIReader* (Java-based DVI file reader). Before highlighting the increasing importance of such a system like *DVIReader*, we will describe the properties of DVI format, and then discuss the features of *DVIReader* by comparing it with related work. Object-Oriented programming and Java technology are introduced in order to help the reader better understand this work. We conclude this chapter by describing the organization of the thesis.

1.1 Basic concept of DVI format

DVI ("**D**evice **I**ndependent") format is the output file format of the TeX typesetting program, designed by David R. Fuchs in 1979 [3]. Unlike the TeX markup files used to generate them, DVI files are not intended to be human-readable; they consist of binary data describing the visual layout of a document in a manner not reliant on any specific image format, display hardware or printer. DVI files are typically used as input to a second program (called a DVI *driver*) which translates DVI files to graphical data. For example, most TeX software packages include a program for previewing DVI files on a user's computer display; this program is a driver. Drivers are also used to convert DVI files to popular document formats (e.g. PostScript and PDF) and for printing. One can also use a PNG driver to generate graphics for mathematical formulae in articles.

DVI is not a document encryption format, and TeX markup may be at least partially reverse-engineered from DVI files, although this process will not to produce high-level constructs identical to those present in the original markup, especially if the original markup used high-level TeX extensions (e.g. LaTeX).

DVI differs from PostScript and PDF in that it does not support any form of font embedding. (Both PostScript and PDF formats can either embed their fonts inside the documents, or reference external ones.) For a DVI file to be printed or even properly previewed, the fonts it references must be supplied. Also, unlike PostScript, DVI is not a full, Turing-complete programming language, though it does use a limited sort of machine language.

Here we would like talk about whether DVI is the best format for binary information exchange or not. This question can probably not be answered definitely without a careful comparison with other formats. The DVI format is interesting at least for following reasons (listed in random order):
a) It allows passage of integer numbers across different CPU architectures.

b) It converts characters through its own conversion table, so that you can use a DVI file generated on an ASCII computer on (say) an EBCDIC computer with no change.

c) A checksum is delivered as a consistency check.

d) A post-amble contains pointers to pages, so that DVI-readers can quickly find a sub-range of pages.

The DVI format was designed to be compact and easily machine-readable. Toward this end, a DVI file is a sequence of commands which form "a machine-like language", in Knuth's words [1]. Each command begins with an eight-bit opcode, followed by zero or more bytes of parameters. For example, an opcode from the group 0x00 through 0x7F (decimal 127), set_char_*i*, typesets a single character and moves the implicit cursor right by that character's width. In contrast, opcode 0xF7 (decimal 247), pre (the preamble, which must be the first opcode in the DVI file), takes at least fourteen bytes of parameters, plus an optional comment of up to 255 bytes.

In a broader sense, a DVI file consists of a preamble, one or more pages, and a post-amble. Six state variables are maintained as a tuple of signed, 32-bit integers: (h,v,w,x,y,z). h and v are the current horizontal and vertical offsets from the upper-left corner (increasing v moves down the page), w and x hold horizontal space values, y and z, vertical. These variables can be pushed or popped from the stack.

Fonts are loaded from PK font files (PK font file means that the font is stored in a compressed, or "packed" format.). The fonts themselves are not embedded in the DVI file, only referenced. Each font, once loaded, is referred to by an internal index for increased compactness of the format.

The DVI format also relies on the character encodings of the fonts it references, not on those of the system processing it. This means, for instance, that

an EBCDIC-based system can process a DVI file that was generated by an ASCII-based system.

This is text and nothing else This is text and nothing else This is text and nothing else This is text and nothing else This is text and nothing else

Scaled Fonts and Standard again,
produces with

```
Scaled \scaledfont Fonts\rm and Standard again,
```

with the LaTeX preamble

```
\newfont{\scaledfont}{cmr12 scaled 7000}
```

Figure 1.1 cmr12 scaled 7000 which looks in DVI

1.2 Object Orientation and Java

Due to the nature of the software development life cycle, constant change is encountered in both functional and non-functional requirements. The key principle of a good software design is the concept of "design for change" [11]. The software development process should have the potential to accommodate as many changes as possible during the course of development and even after the system in use. So the software design goals should include high cohesion of functions within a module, loose coupling of functions between modules and, in addition, the design should be carefully abstracted and easy to understand.

In object oriented programming, a model is created for a real world system. Classes are programmer-defined types that model the components of the system. An object is a software bundle of related variables and methods. Using the prototype provided by a class, the programmer can create a number of objects, each of which is called an instance of the class. Different objects of the same class have the same fields and methods, but the values of the fields will in general differ. Software objects interact with one another by exchanging messages. A class can inherit state and behavior from its super-class. Inheritance provides a powerful and natural mechanism for organizing and structuring software programs [11].

DIVReader was implemented using JAVA application technology. JAVA technology is both a programming language and a platform. The Java language is an object-oriented programming language developed by Sun Microsystems [11]. It has become one of the most popular programming languages due to a number of features. Sun Microsystems describes Java as simple, object-oriented, distributed, robust, and secure, and dynamic language [11, 12]. The Java platform differs from most other platforms in that it is a software-only platform that runs on top of the other hardware-based platforms. The *Java Virtual Machine* (JVM) is the base of the Java platform and can be implanted into various hardware-based

platforms. In other words, it means that as long as a computer has a JVM, the same program written in Java programming language can run on different operating systems such Windows, or Solaris, with identical results.

1.3 Overview and Requirements of DVIReader

DVIReader is a tool for previewing DVI files, so its main application is viewing the typeset results while proofreading and printing a (La)TeX file. The *DVIReader* application is also a tool that makes user's life easy for those who try to browse the DVI files in the Windows platform.

1.3.1 Working process

The following section is the working process of the tool demonstrating its workflow sequence from input through output.

1. Input:
    1. A standard DVI file
    2. Encapsulated postscript files referenced by the DVI file
    3. A template HTML file
2. Available Options
    1. Page numbering
    2. Scale factors
    3. Layout and color
    4. Name of a template HTML file
    5. PK Font auto-selection
3. Output
    1. Smaller (1 page) DVI files for faster loading
    2. GIF files for each image and scale factor

DVIReaderDocument and *DVIReaderPanel* modules are both the most visible and the most complicated part of the system.

1. *DVIReaderPanel* displays one page from an ordinary DVI file
    1. Loads, decompresses, and scales PK fonts
    2. Looks for GIF files corresponding to eps file format specials
    3. Understands internal and external links of HTML
2. Have navigation controls (Zoom, print, etc.)
3. Manages multiple views of a single file, including the magnifier window

1.3.2 Standard File Formats Required

1. The viewing panel should load standard DVI files and PK font files.

   These file formats are remarkably compact, and so work well on the Windows platform. Experimentation shows that they can be decoded quickly in Java.

   We experimented with decoding, scaling, and colorizing PK fonts, also we tried zipping the font file into a Jar file, but it was taking much more time to process, and then we decided that put those PK font files in locally.

2. Any special files produced to improve performance should also give every appearance of being standard DVI or PK files.

   The one-page DVI files which are produced refer to each other by their DVI file names; *DVIReader* figures out which html page is loaded to traverse a link. These files also retain their original ps, GIF or eps files specials (string \specials which can be interpreted by the viewer as hypertext, EPS, GIF, etc.); the software figures out which file is loaded to display the image on pre-retained space.

1.3.3 Flexibility

1. The system supports display within a standalone application.
2. The system supports flexible data formats.

   It is easy to add support for a new special or for another font file format.

1.4 Related Work

There are several tools implemented to view DVI format files in Windows platform such as Xdvi and IDVI etc. Xdvi and IDVI will be introduced in this section [7, 17].

Xdvi is a tool for previewing DVI files, so its main application is viewing the typeset results while editing or proofreading a (La)TeX file. It is not targeted at doing presentations or multimedia.

Xdvi is built on an architecture that separates functionality from concrete document format. Almost all functionality is made available via relatively small modules of code called behaviors that programmers can write to extend the core system [7]. Behaviors can be as significant and powerful as parser-renders for scanned paper, HTML, or TeX DVI; as fine-grained as hyperlinks, cookies, and the disabling of menu items; and as innovative or uncommon as in situ annotations, "lenses", collapsible outline displays, new GUI widgets, and Robust Hyperlink support. Behaviors can be combined in arbitrary groups for each individual document, in effect spontaneously creating a custom browser for every one. Common aspects of document functionality can be shared, so that, for example, the same behavior that handles multi-page support for scanned paper documents also provides such support for DVI and PDF; similarly, the behaviors that support fine-grain annotation of HTML also support identical annotation on scanned paper, UNIX manual pages, DVI, and PDF.

IDVI is also a tool for previewing DVI files, in additional, it allows the users to present documents on the web. IDVI design goals will take advantage of the browser, motivates a design where state changes are achieved by loading a new document. Then the user can undo the change by using the back button, and save all state by placing a bookmark. The page number and magnification level are handled this way. Also IDVI try to make DVI files act like other browser content.

But even with this limited application range, there is still a lot of room for improvement. After we were using *DVIReader* to compare with Xdvi and IDVI, we found that Xdvi is the closest system that we have come across. In both *DVIReader* and Xdvi systems, the DVI-format is the basic structuring component and the algorithms are objectified as separate objects.

Xdvi and IDVI are not targeted at doing presentations or multimedia. Also, in contrast to Xdvi and IDVI, we chose Java Patterns for the implementation of *DVIRreader*, for example, Fast Synchronized Initialization, object sharing and Group Locking are used in our system.

Fast Synchronized Initialization

It is often the case that we don't want to construct an object until it is first needed. For example, after loading the font file *cmr12.7000pk*, we may not want to immediately decompress the character data for each character of that font. The reason is that when you first encounter a character in a new font, you don't know how many other characters from that font will be needed, perhaps no more characters will be needed, or maybe the whole font will be used. In the later case, it will take too much time and memory, and it is unlikely that every character will be needed right away.

The natural way to delay construction is to store a NULL pointer in place of the object, and test for NULL before using the object:

```
static Object object;
```

```
void useObject( ) {
    if( object == NULL )
        object = new Object( );

    // use object now
}
```

This scheme fails when there may be multiple threads accessing the object. This may occur if an object is referenced from more than one document, or if, as in *DVIREADER*, documents are being loaded by separate threads.

The obvious solution is to make the useObject method synchronized:

```
static Object object;

synchronized void useObject( ) {
    if( object == NULL )
        object = new Object( );

  // use object now
}
```

Synchronization is slow (5-10 μs, even when running inside a JIT environment) [12, 16]. If you might need to call useObject or methods like it thousands of times per second, then the overhead is quite large. The trick is to avoid synchronization once the object has been constructed, while still using a synchronized method to prevent multiple copies from being constructed:

```
static Object object;

void useObject( ) {
    if( object == NULL )
```

```
            constructObject( );

        // use object now
    }

    synchronized void constructObject( )
    {
        if( object == NULL )
            object = new Object( );
     }
```

This idiom is pervasive in *DVIREADER*. It is used for constructing decompressed character data, for creating Image objects to represent individual characters, and for creating Color objects used to represent the colors used in drawing a rule, for example.

The idiom is used when an object may or may not ever need to be constructed, but may be used many times once it is constructed. Decompressed character data is probably the best example. If a symbol appears once in a document, it will likely appear many times.

Object Sharing

An object cache can be implemented as static Hashtable member of a class. Access is through a synchronized static method, which creates the Hashtable if it has not yet been created, and creates requested objects if they are not already in the Hashtable:

```
Hashtable documentCache;

 synchronized DVIDocument getDocument( URL
 documentURL ) {
```

```
    if( documentCache == NULL )
        documentCache = new Hashtable( );

    DVIDocument result = documentCache.get(
documentURL );

    if( result == NULL ) {
        result = new DVIDocument( documentURL );
        documentCache.put( documentURL, result );
    }

    return result;
    }
```

The Hashtable should not be initialized where it is defined, since the static initialization method is not synchronized by default. This can cause repeated re-initialization of static objects.

The fast synchronized initialization scheme should not be used for constructing the hashtable, since we need to synchronize the construction of the object being retrieved as well.

This idiom is used whenever there should be a unique object of a class for each set of parameters. For example, it is used in the construction of DVIDocument objects (one per document URL), DVIFont objects (one per font name and size).

Group Locking

The Java language provides a single monitor per object of a given class, plus another for the class itself. It is tempting to use this monitor directly for all synchronization that needs to occur in a Java program.

The classical solution for deadlock is to ensure that locks are always acquired in a particular order. In this case, either the top-down or bottom-up traversals need to be made unsynchronized. This may not be practical.

Another solution is to add a second monitor to each object using an explicit Lock object, and use one set for top-down and the other set for bottom-up traversal. This can be made to work, but is difficult and error prone.

The solution used in *DVIREADER* is a vastly simplified one, but one which was certainly not obvious at first: use a single ReadWriteLock to control access to the entire hierarchy. As long as individual accesses are relatively fine-grained, this provides sufficient opportunity for multi-threading while reducing complexity and eliminating the possibility of deadlock.

In general, we use a single lock to control access to a large complicated data structure, instead of synchronizing access to each part of the data structure.

1.5 Organization

In the next chapter, we present the background of LaTex; provide documentation for the DVI file format in this chapter; and discuss system functional descriptions for each module. Chapter 3 introduces our proposed objected-oriented design of *DVIReader* and describes the dynamic behavior of each module. In Chapter 4, we present a testing plan and results to illustrate how we can use *DVIReader* to actively read a DVI file. Chapter 5 concludes the thesis with discussion and future research directions.

Chapter 2 LaTeX and the DVI File Format

TeX plays a very important role in scientific and technical publishing. Thus far it has however been almost exclusively confined to the production of a paper end product [4].

2.1 Getting Start with LaTeX

2.1.1 TeX and LaTeX

TeX is a computer program for typesetting documents, created by D. E. Knuth [4, 5, 6]. It takes a suitably prepared computer file and converts it to a form that may be printed on many kinds of printers, including dot-matrix printers, laser printers and high-resolution typesetting machines. A number of well-established publishers now use TeX in order to typeset books and mathematical journals.

Simple documents that do not contain mathematical formulae or tables may be produced very easily: the body of the text is typed in essentially unaltered (though the user must observe certain rules regarding quotation marks and punctuation dashes). Typesetting mathematics is somewhat more involved, but even here TeX is comparatively straightforward to use when one considers the complexity of some of the formulae that it is required to typeset.

LaTeX, written by L. B. Lamport, is one of a number of "dialects" of TeX [4,5,6,13,14]. It is particularly suited to the production of long articles and books, since it has facilities for the automatic numbering of chapters, sections, theorems, equations etc., and also has facilities for cross-referencing. It is probably one of the most suitable versions of TeX for beginners to use. LaTeX is a high-quality typesetting system, with features designed for the production of technical and

scientific documentation. LaTeX is the *de facto* standard for the communication and publication of scientific documents.

2.1.2 A Typical LaTeX Input File

The LaTeX program reads in text from a suitably prepared input file, and creates a "DVI file" which encodes information on the fonts to be used and the positioning of the characters on the printed page. There are many programs available that can translate the "DVI file" into page description languages such as "PostScript", or convert it into the format appropriate for previewing the document on a computer screen or printing it out on dot-matrix printers.

Here is an example of a typical LaTeX input file:

```
\documentclass[a4paper,12pt]{article}
\begin{document}

The foundations of the rigorous study of
\emph{analysis}
were laid in the nineteenth century, notably by the
mathematicians Cauchy and Weierstrass. Central to the
study of this subject are the formal definitions of
\emph{limits} and \emph{continuity}.

Let $D$ be a subset of $\bf R$ and let
$f \colon D \to \mathbf{R}$ be a real-valued function
on
$D$. The function $f$ is said to be \emph{continuous}
on
$D$ if, for all $\epsilon > 0$ and for all $x \in D$,
there exists some $\delta > 0$ (which may depend on
$x$)
such that if $y \in D$ satisfies
\[ |y - x| < \delta \]
then
\[ |f(y) - f(x)| < \epsilon. \]

One may readily verify that if $f$ and $g$ are
continuous
functions on $D$ then the functions $f+g$, $f-g$ and
$f.g$ are continuous. If in addition $g$ is
everywhere
```

```
non-zero then $f/g$ is continuous.

\end{document}
```

When we apply LaTeX to these paragraphs we produce the text

The foundations of the rigorous study of *analysis* were laid in the nineteenth century, notably by the mathematicians Cauchy and Weierstrass. Central to the study of this subject are the formal definitions of *limits* and *continuity*.

Let D be a subset of $\mathbf{R}$ and let $f\colon D \to \mathbf{R}$ be a real-valued function on D. The function f is said to be *continuous* on D if, for all $\epsilon > 0$ and for all $x \in D$, there exists some $\delta > 0$ (which may depend on x) such that if $y \in D$ satisfies

$$|y - x| < \delta$$

then

$$|f(y) - f(x)| < \epsilon.$$

One may readily verify that if f and g are continuous functions on D then the functions $f + g$, $f - g$ and $f.g$ are continuous. If in addition g is everywhere non-zero then f/g is continuous.

Figure 2.1 Text View in LaTex

This example illustrates various features of LaTeX. Note that the lines

```
\documentclass[a4paper,12pt]{article}
\begin{document}
```

are placed at the beginning of the input file. These are followed by the main body of the text, followed by the concluding line

```
\end{document}
```

Note also that, although most characters occurring in this file have their usual meaning, there are characters such as \, $, { and } that have special meanings within LaTeX. In particular, there are sequences of characters which begin with a “backslash” \ which are used to produce mathematical symbols and Greek letters and to accomplish tasks such as changing fonts. These are known as control sequences.

2.1.3 Characters and Control Sequences

Most characters on the keyboard, such as letters and numbers, have their usual meaning. However the characters `\ { } $ ^ _ % ~ # &` are used for special purposes within LaTeX. Thus typing one of these characters will not produce the corresponding character in the final document. Of course these characters are very rarely used in ordinary text, and there are methods of producing them when they are required in the final document.

In order to typeset a mathematical document it is necessary to produce a considerable number of special mathematical symbols, and to change fonts where appropriate. Mathematical documents often contain arrays of numbers or symbols (matrices) and other complicated expressions.

These are produced in LaTeX using *control sequences*. Most control sequences consist of a backslash \ followed by a string of (upper or lower case) letters. For example, `\delta, \emph and \to` are control sequences: the control sequence `\delta` produces the Greek letter δ, the control sequence `\emph`, when followed by text enclosed within braces, will cause that text to be emphasized (usually by typesetting it in an *italic font*), and the control sequence `\to` (or `\rightarrow`) produces the arrow $\rightarrow$.

There is another type of control sequence that consists of a backslash followed by a single character that is not a letter. Examples of control sequences of this type are `\{, \" and \$.`

The “braces” { and } are used for grouping: the characters they enclose are treated as a single “group”, which can be specified as an “argument” of a control sequence such as `\emph`, or as a superscript or subscript in a mathematical formula. Control sequences included in such a group apply only to the contents of the group.

The special character $ is used when embedding mathematical expressions in paragraphs of ordinary text in order to change into and out of “mathematics mode”.

The special characters ^ and _ are used in mathematical expressions to produce superscripts and subscripts respectively.

The special character % is used to introduce "comments" into the input file that do not appear in the final document: all characters occurring after % on any line of the input file are ignored by LaTeX.

The special character # is used to specify arguments in definitions of control sequences. The special character & is used when typesetting tables in order to separate entries in different columns.

2.2 Documentation for the DVI file format

This purpose of this paragraph is to provide documentation for the DVI file format.

2.2.1 The DVI File Format

Before we get into the details of DVI type, we need to know exactly what DVI files are. The form of such files was designed by David R. Fuchs in 1979 [1, 3]. Almost any reasonable typesetting device can be driven by a program that takes DVI files as input, and dozens of such DVI-to-whatever programs have been written. Thus, it is possible to print the output of document compilers like TeX on many different kinds of equipment.

A DVI file is a stream of 8-bit bytes, which may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the `set_rule` command has two parameters,

each of which is four bytes long. Parameters are usually regarded as nonnegative integers; but four-byte-long parameters, and shorter parameters that denote distances, can be either positive or negative. Such parameters are given in two's complement notation. For example, a two-byte-long distance parameter has a value between -2^15 and 2^15-1. **[NOTE: DVI files use big endian format for multiple byte integer parameters.]**

A DVI file consists of a "preamble", followed by a sequence of one or more "pages", followed by a "postamble". The preamble is simply a `pre` command, with its parameters that define the dimensions used in the file; this must come first. Each "page" consists of a `bop` command, followed by any number of other commands that tell where characters are to be placed on a physical page, followed by an `eop` command. The pages appear in the order that they were generated, not in any particular numerical order. If we ignore `nop` commands and `fnt_def` commands (which are allowed between any two commands in the file), each `eop` command is immediately followed by a `bop` command, or by a `post` command; in the latter case, there are no more pages in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in DVI commands are "pointers". These are four-byte quantities that give the location number of some other byte in the file; the first byte is number 0, then comes number 1, and so on. For example, one of the parameters of a `bop` command points to the previous `bop`; this makes it feasible to read the pages in backwards order, in case the results are being directed to a device that stacks its output face up. Suppose the preamble of a DVI file occupies bytes 0 to 99. Now if the first page occupies bytes 100 to 999, say, and if the second page occupies bytes 1000 to 1999, then the `bop` that starts in byte 1000 points to 100 and the `bop` that starts in byte 2000 points to 1000. (The very first bop, i.e., the one that starts in byte 100, has a pointer of -1.)

The DVI format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information implicit instead of explicit. When a DVI-reading program reads the commands for a page, it keeps track of several quantities: (a) The current font f is an integer; this value is changed only by `fnt` and `fnt_num` commands. (b) The current position on the page is given by two numbers called the horizontal and vertical coordinates, h and v. Both coordinates are zero at the upper left corner of the page; moving to the right corresponds to increasing the horizontal coordinate, and moving down corresponds to increasing the vertical coordinate. Thus, the coordinates are essentially Cartesian, except that vertical directions are flipped; the Cartesian version of `(h,v)` would be `(h,-v)`. (c) The current spacing amounts are given by four numbers w, x, y, and z, where w and x are used for horizontal spacing and where y and z are used for vertical spacing. (d) There is a stack containing `(h,v,w,x,y,z)` values; the DVI commands push and pop are used to change the current level of operation. Note that the current font f is not pushed and popped; the stack contains only information about positioning.

The values of `h, v, w, x,` y, and z are signed integers having up to 32 bits, including the sign. Since they represent physical distances, there is a small unit of measurement such that increasing h by 1 means moving a certain tiny distance to the right. The actual unit of measurement is variable, as explained below.

2.2.2 DVI Format

This following section demonstrates a description of dvi format regarded as a series of commands in a machine-like language [1, 2].

Field	Size	Description
no_ops	>= 0 bytes	(NOP, nops before the preamble)
preamble_marker	1 ubyte	(PRE)
version_id	1 ubyte	(should be version 2)
numerator	4 ubytes	(numerator must equal the one in postamble)
denominator	4 ubytes	(denominator must equal the one in postamble)
magnification	4 ubytes	(magnification must equal the one in postamble)
id_len	1 ubyte	(length of identification string)
id_string	id_len ubytes	(identification string)
no_ops	>= 0 bytes	(NOP, nops before a page)
begin_of_page	1 ubyte	(BOP)
page_nr	4 sbytes	(page number)
do_be_do	36 bytes	(filler)
prev_page_offset	4 sbytes	(offset in file where previous page starts, -1 for none)
end_of_page	1 ubyte	(EOP)
no_ops	>= 0 bytes	(NOPS)
postamble_marker	1 ubyte	(POST)
last_page_offset	4 sbytes	(offset in file where last page starts)
numerator	4 ubytes	(numerator must equal the one in preamble)
denominator	4 ubytes	(denominator must equal the one in preamble)
magnification	4 ubytes	(magnification must equal the one in preamble)
max_page_height	4 ubytes	(maximum page height)
max_page_width	4 ubytes	(maximum page width)
max_stack	2 ubytes	(maximum stack depth needed)
total_pages	2 ubytes	(number of pages in file)
postamble_offset	4 sbytes	(offset in file where postamble starts)
version_id	1 ubyte	(should be version 2)
trailer	>= 4 ubytes	(TRAILER)

```
FONT DEFINITIONS:
  do {
    switch (c = getc(dvi_fp) {
```

```
      case FNTDEF1  :
      case FNTDEF2  :
      case FNTDEF3  :
      case FNTDEF4  : define_font(c);
      case POSTPOST : break;
      default       : error;
    }
  } while (c != POSTPOST);
```

2.2.3 Table of Opcodes

The following table gives the instruction set for DVI. The parameters are listed in the order they would appear in a DVI file; the number in brackets gives the size of the parameter (in bytes). We put the description of the Opcodes into Appendix A [3].

The DVI Instruction Set

Opcode	Instruction Name	Parameters	Description
0...127	set_char_*i*		typeset a character and move right
128	set1	c[1]	
129	set2	c[2]	typeset a character and move right
130	set3	c[3]	
131	set4	c[4]	
132	set_rule	a[4], b[4]	typeset a rule and move right
133	put1	c[1]	
134	put2	c[2]	typeset a character
135	put3	c[3]	
136	put4	c[4]	
137	put_rule	a[4], b[4]	typeset a rule

138	nop		no operation
139	bop	c_0[4]..c_9[4], p[4]	beginning of page
140	eop		ending of page
141	push		save the current positions
142	pop		restore previous positions
143	right1	b[1]	
144	right2	b[2]	move right
145	right3	b[3]	
146	right4	b[4]	
147	w0		move right by w
148	w1	b[1]	
149	w2	b[2]	move right and set w
150	w3	b[3]	
151	w4	b[4]	
152	x0		move right by x
153	x1	b[1]	
154	x2	b[2]	move right and set x
155	x3	b[3]	
156	x4	b[4]	
157	down1	a[1]	
158	down2	a[2]	move down
159	down3	a[3]	
160	down4	a[4]	
161	y0		move down by y
162	y1	a[1]	
163	y2	a[2]	move down and set y
164	y3	a[3]	
165	y4	a[4]	
166	z0		move down by z
167	z1	a[1]	
168	z2	a[2]	move down and set z
169	z3	a[3]	
170	z4	a[4]	
171...234	fnt_num_*i*		set current font to i
235	fnt1	k[1]	
236	fnt2	k[2]	set current font
237	fnt3	k[3]	
238	fnt4	k[4]	

239	xxx1	k[1], x[k]	
240	xxx2	k[2], x[k]	extension to DVI primitives
241	xxx3	k[3], x[k]	
242	xxx4	k[4], x[k]	
243	fnt_def1	k[1], c[4], s[4], d[4], a[1], l[1], n[a+l]	
244	fnt_def2	k[2], c[4], s[4], d[4], a[1], l[1], n[a+l]	define the meaning of a font number
245	fnt_def3	k[3], c[4], s[4], d[4], a[1], l[1], n[a+l]	
246	fnt_def4	k[4], c[4], s[4], d[4], a[1], l[1], n[a+l]	
247	pre	i[1], num[4], den[4], mag[4], k[1], x[k]	preamble
248	post	p[4], num[4], den[4], mag[4], l[4], u[4], s[2], t[2] <*font definitions*>	postamble beginning
249	post_post	q[4], i[1]; 223's	postamble ending
250...255	*undefined*		

Table 2.1 Tables of Opcodes

2.3 User Interface Design

A user friendly GUI (Graphic User Interface) is required for the user to use the tool. The user interface will be simple, clear and easy to run. The main requirements for the user interface are as follows:

- A text field is required for user to view the DVI files
- A menu field is required for the user to select the different functionality such as "open a dvi file" or "change the text color".
- Lists of buttons are required for the user to control the text field easily.

2.4 Module Description

Font module:

The static method *DVIFont.LoadFont* is used to load fonts. Given a font description, it returns an object of a class derived from *DVIFont*. This object must be able to provide an array of `0/1` values representing a given character from the font, as well as information about the dimensions of the character.

DVIREADER requires that all fonts used in a document are available as pk font files at the correct sizes. If they aren't available, it searches for a nearby size, and fails if it can't find one.

***DVIReaderDocument* and *DVIReaderPanel* module:**

Those modules are both the most visible and the most complicated part of the system. ***DVIReaderPanel*** displays one page from an ordinary DVI file, the viewing panel load standard DVI files and PK font files. These file formats are remarkably compact, and so work well on the Windows platform. Experimentation shows that they can be decoded quickly in Java. The following are the major functions of these two modules.

1. Loads, decompresses, and scales PK fonts
2. Looks for GIF files corresponding to eps links
3. Understands internal and external links
4. Has navigation controls (Zoom, print, etc.)
5. Manages multiple views of a single file, including the magnifier window

Chapter 3 Objected-Oriented Design and Implementation of DVIReader

As discussed in Chapter 2, I believe that understanding the composition of a DVI file is a challenging task since the application tool should figure out DVI format not only during the design but also at run time. With this viewpoint, we now introduce the object-oriented design of *DVIReader* in detail. We first present the overview of *DVIReader* system architecture, and describe how the framework integrates with each function. Then we give an in-depth explanation of the proposed modular design. Also we will describe algorithms of major functions. Finally, we illustrate the dynamic behavior of each module through a set of sequence diagrams.

3.1 System Architecture

Figure 3.1 presents an overview of object-oriented system architecture design of *DVIReader*. It includes three layers: User Interface, functions controller and Font decoder. The user interface provides for a user request to open a DVI format file. The tool contains three major modules: *DVIReaderMainWindow* Module, *DVIReader Panel* Module, and *DVIReader* Document Module.

When the user selects the needed DVI file from the user interface, the URL of selected file is sent to the *DVIReaderMainWindow* for processing. The *DVIReaderMainWindow* module passes the request to the *DVIReaderPanel* module to load, decompress, and scale PK fonts, and locates the font name. The request is then passed to the font module to process and it will parse the matched PK fonts for displaying the result. If they are not available, *DVIReaderPanel* Module searches for a nearby size, and fails if it can not find one.

In a similar way, *DVIReader* Document Module looks for GIF files corresponding to eps links and understands internal and external links handles zoom in, out, prints controls when processing DVI file at same time.

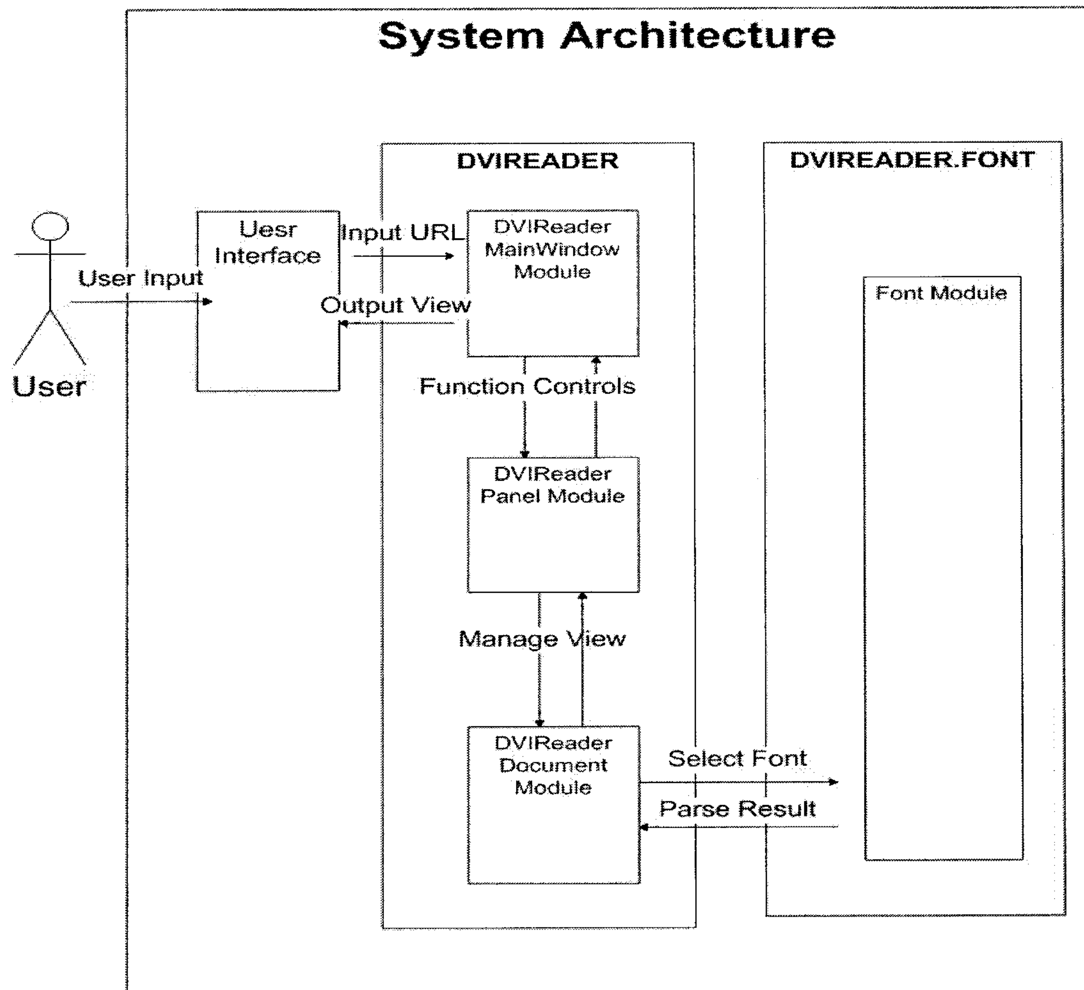

Figure 3-1 Architecture Design Diagram

3.2 In-Depth View of Design

As mentioned, *DVIReader* is fully object-oriented. Every component in the system is objectified without any concerns of it being a user interface or the function abstraction. In the subsequent sections, we discuss the detailed design and the implementations for those components to collaborate together.

In Figure 3.3, a class diagram of the system shows nine main classes involved in the system. We designed the nine classes as follows:

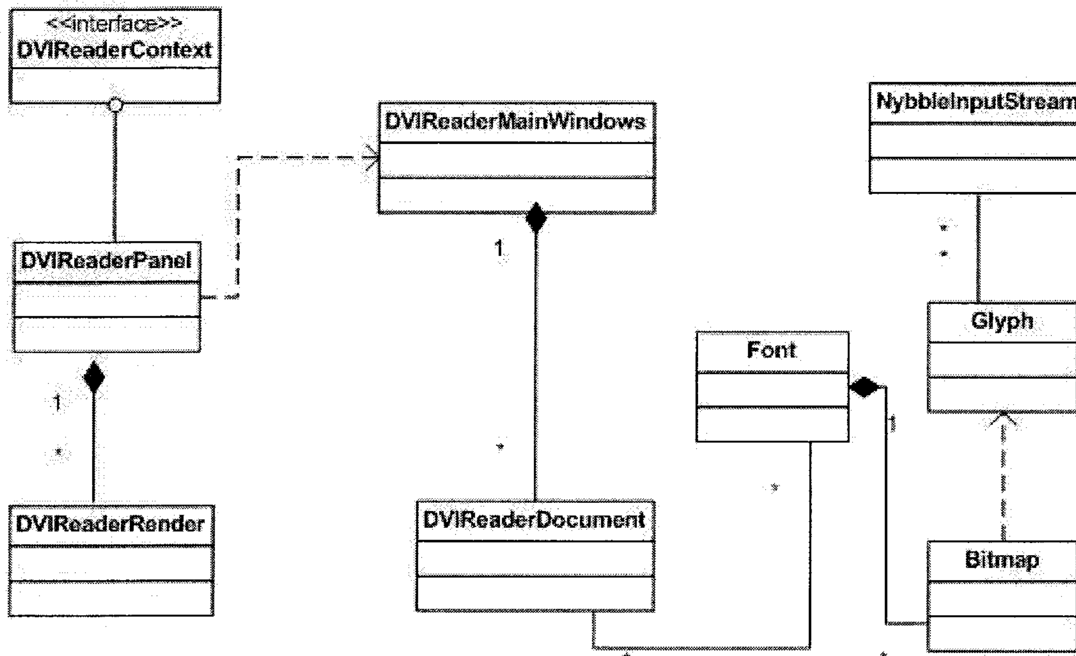

Figure 3.3 System Class Diagram

DVIReaderMainWinow module provides management for the DVI Controls window. A thread is used to hide the window after the user leaves a DVI page. This ensures that if the user is leaving one DVI page on their way to another, the controls window stays up. Instance functions manage the construction of the DVI Controls window, forward messages to the window, and manage the magnified view. There are also panels that maintain the navigation buttons and menus at the top of the DVI Controls window. The custom layout manager *awt.RegularLayout* is used to control the positioning of the buttons.

Interface *DVIReaderContext* describes the methods *DVIReaderPanel* needs for interacting with its surrounding. *DVIReaderPanel* assumes that it has got an implementation of this interface set with *setDVIReaderContext()*;

The *DVIReaderPanel* module is the central function of *DVIReader;* it does all the housekeeping, etc. In particular, it triggers the *DVIReaderRender* class and shows a DVI file. It describes the format of a DVI file, and the format of the various specials that are recognized, and it also provides some utilities for derived classes, such as font management functions and a function to parse the header of a DVI file. The first step in loading a page is to create an empty Block hierarchy, and return this to the caller. This can be added to a set of pages, and *ViewPanel* objects can be created which display the (empty) Block hierarchy. Later, the caller asks that the page actually be loaded. If any *ViewPanels* displaying the hierarchy have been created, then the loading process will generate repaint requests and the block hierarchy will be displayed as it is being loaded.

The *DVIReaderRender* class hides all information about device-independent coordinates and magnification levels. All externally visible coordinates are integers, at the un-scaled resolution specified for PK font files. It is requires a *DVIReaderDocument* object, which manages fonts.

The *DVIReaderDocument* module stores all document wide information. It loads a document from a URL and loads asynchronously, so that a request for a certain page may block. Also it loads fonts as they are defined in the body of the DVI file. It may use an object to convert a token stream into one object per page. Also it would read the post-amble in order to define all fonts right away, and scan backwards through the document getting page offsets; then load a page only when it is requested for the first time.

The *Bitmap* class is used to store a bitmap (or a bit array) efficiently. Moreover it can generate an image from the bitmap that is scaled with respect to a given point. This is important for *fonts* since it ensures that the baseline gets aligned properly.

The *Font* class encapsulates the information from a PK font file. It knows how to parse a PK file. It loads and decodes a PK font file asynchronously, so that a request for a certain character may block. It will creates *Glyph* objects to decode and represent individual characters from the font.

The *Glyph* class represents a single character from a PK font file. It knows how to generate an image of a scaled instance of itself, which makes it easier to read individual bits from a *NybbleInputStream*.

The *NybbleInputStream* class encapsulates an input stream and provides methods to read different length values from nybbles to 4 byte ints as needed in parsing .pk files, which makes it easier to read individual nybbles from a *InputStream*.

In *DVIReader*, the objects in the system perform task independently and the interactions between them are easily handled. This dependent design enables each module to be easily implemented and/or extended in the future.

3.3 Detailed Design Description

This section describes the user interface design, and then shows the dynamic behavior among objects in each module with the help of sequence diagrams. A sequence diagram, on the horizontal axis, shows the lifetime of the object that it represents, while on the vertical axis, shows the sequence of creation or invocation of these objects [10,15]. A sequence diagram clearly depicts the sequence of events, and allows the designer to specify the sequence of messages sent between objects in collaboration over time. It is an excellent tool for depicting concurrent operations in each module of *DVIReader*. Also we will describe major Class Diagram and some Key algorithms in this system at the beginning.

3.3.1 User Interface Design

The *java.awt* package is used in this system to create the user interface. The System user interface is dynamically generated from the objects *Service*, *Category*, *Enum* , *Filter* and *Range*, which are constructed from an input-stream. Grid-layout is used to partition the main window for the categories. There is a description area at the bottom of the window.

Figure 3.3 shows Windows Main Page. It contains a menu bar, a tool bar, and an output text field.

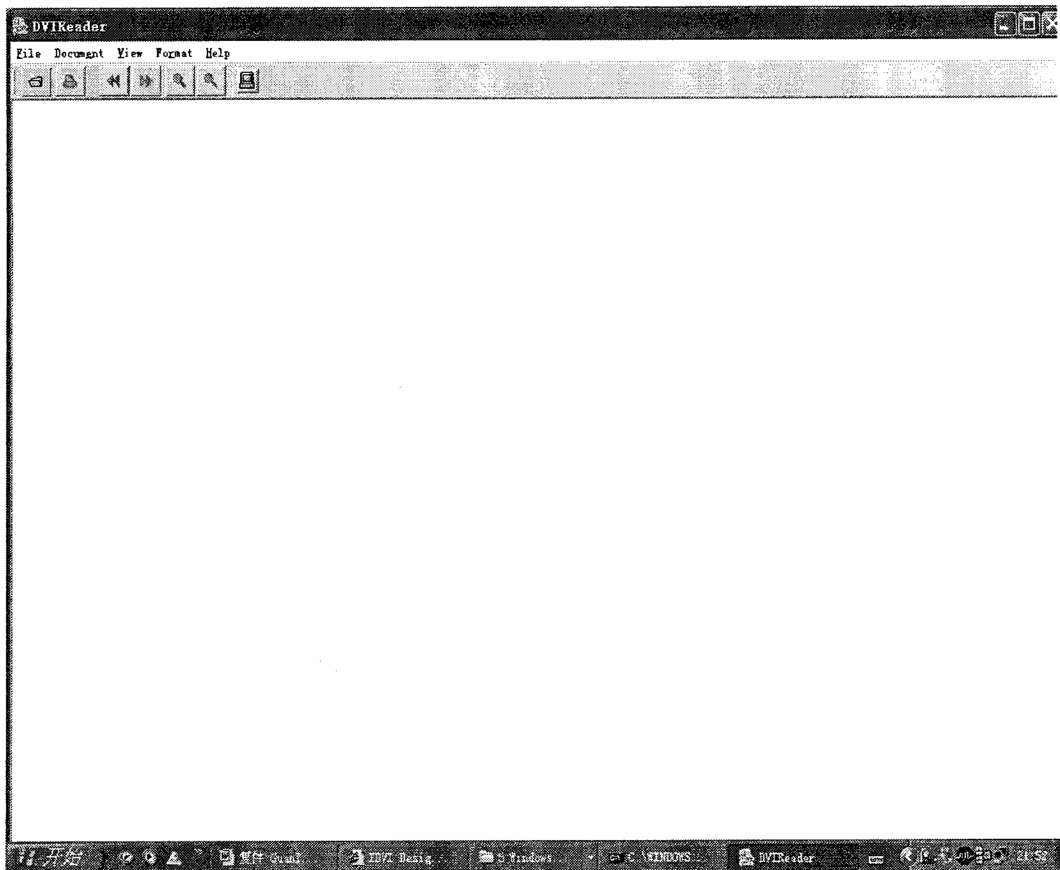

Figure 3.3 Main Window

Figure 3.4 and Figure 3.5 show the result Windows when the user selects a DVI file.

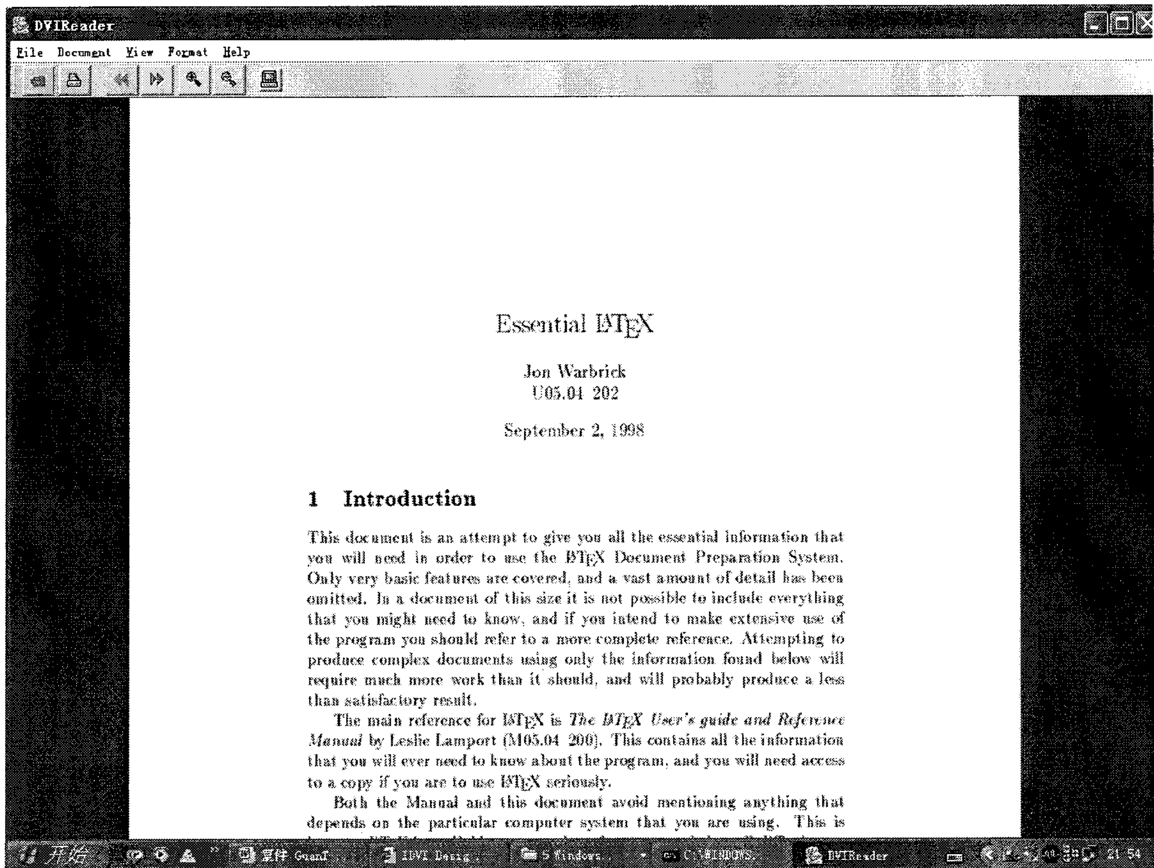

Figure 3.4 Result Window 1

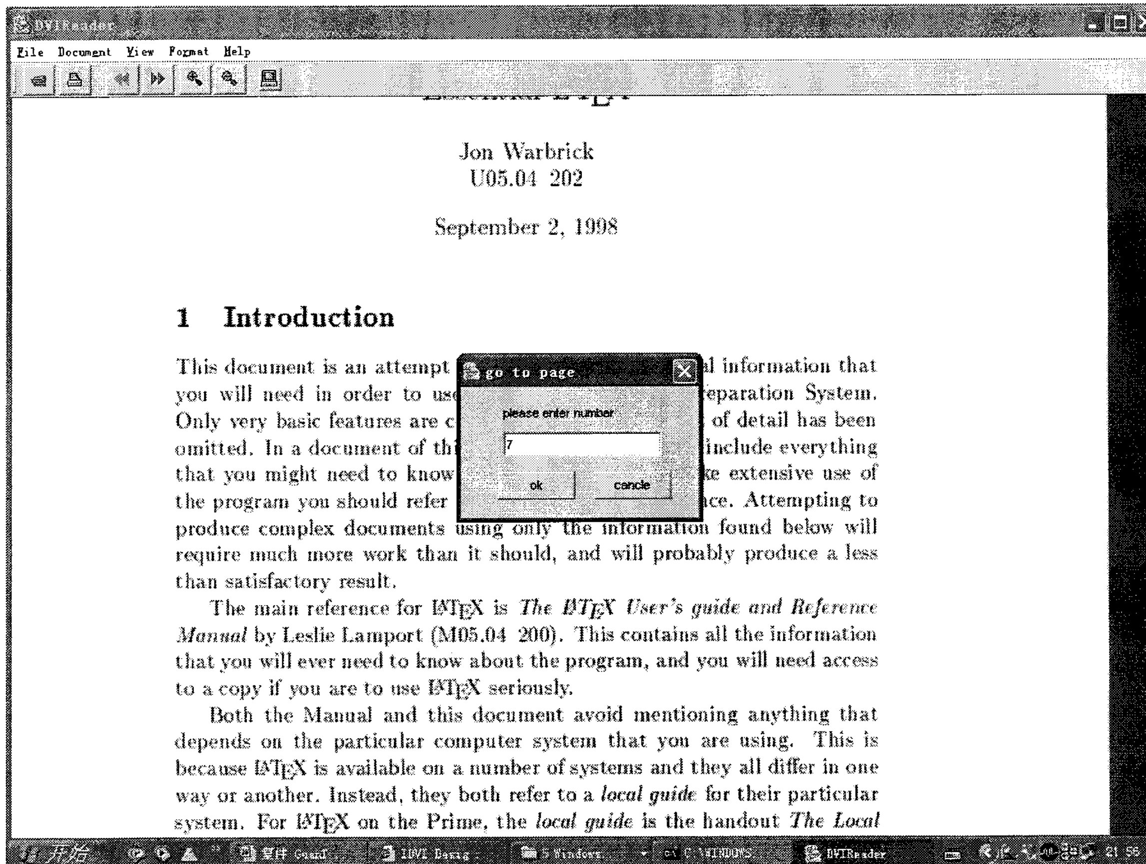

Figure 3.5 Result Window 2

3.3.2 Major Class Diagram and Key algorithms

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system: conceptual, specification, and implementation.

The **MainWindow Class** provides management for the DVI Controls window. This class is the main class for the system. It will create the user interface and the major executive functions that connect the interface with the event handler. Figure 3.6 is the UML presentation of the class.

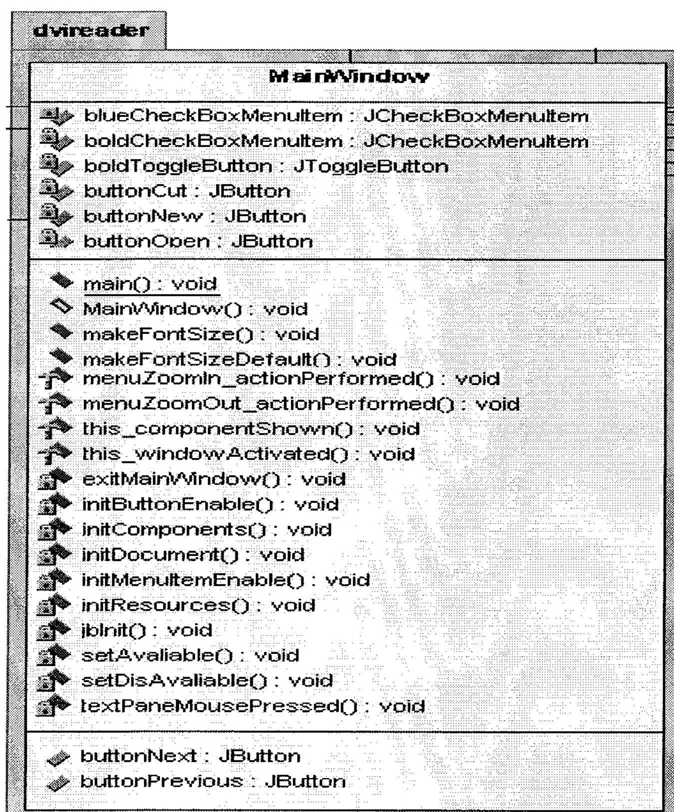

Figure 3.6 MainWindow Class

The ***DVIReaderContext*** **Interface Class** describes the methods that *DVIReaderPanel* needs for interacting with its surrounding. Figure 3.7 is the UML presentation of the class.

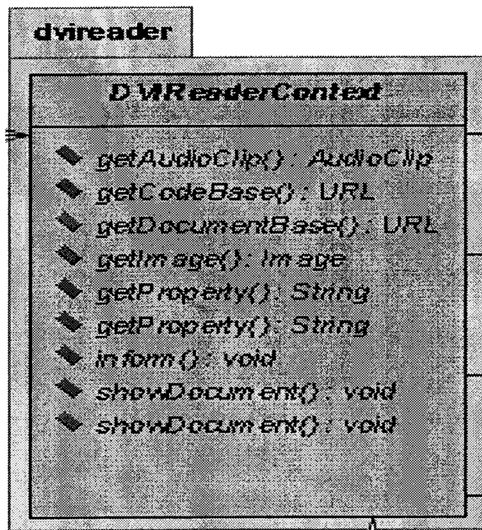

Figure 3.7 *DVIReaderContext* Interface Class

The ***DVIReaderPanel*** **Class** triggers the *render* object and shows a DVI file. It provides page management functions and functions to show status of a DVI file, such as *showDocument(), zoomByScale()*, etc,. Figure 3.8 is the UML presentation of the class.

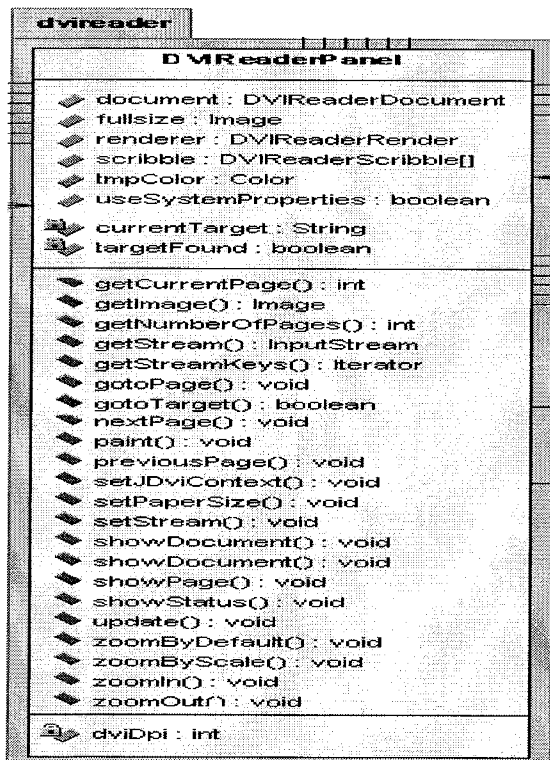

Figure 3.8 DVIReaderPanel Class

The ***DVIReaderDocument*** **Class** stores all document wide information. The major functions in this class are to load fonts as they are defined in the body of the DVI file, scan the page and parse to find the preamble, post-amble. Method *parsePre()* and *parsePost()* are the key functions. Figure 3.9 is the UML presentation of the class.

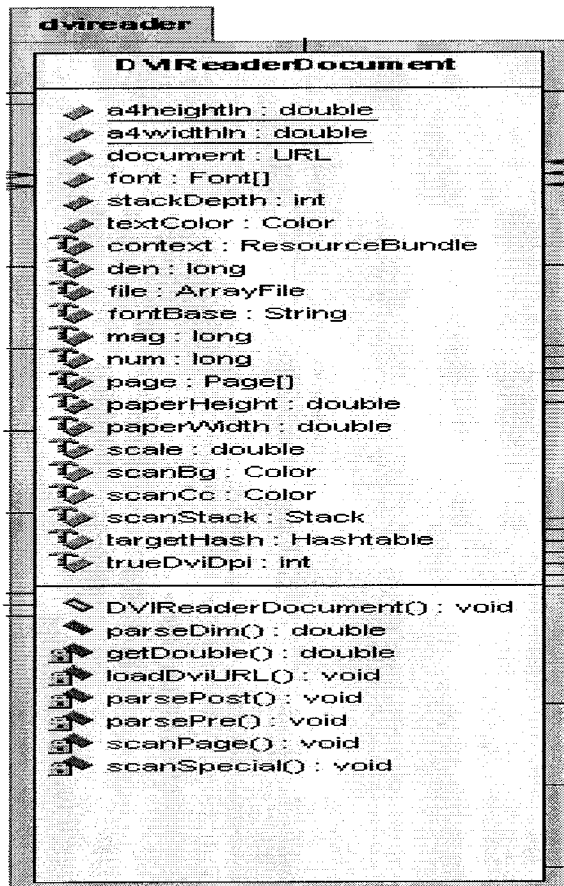

Figure 3.9 ***DVIReaderDocument*** **Class**

The ***DVIReaderRender*** **Class** hides all information about device-independent coordinates and magnification levels. All externally visible coordinates are integers, at the un-scaled resolution specified for PK font files. Method *renderPage(), flushFonts()* are the key functions. Figure 3.10 is the UML presentation of the class.

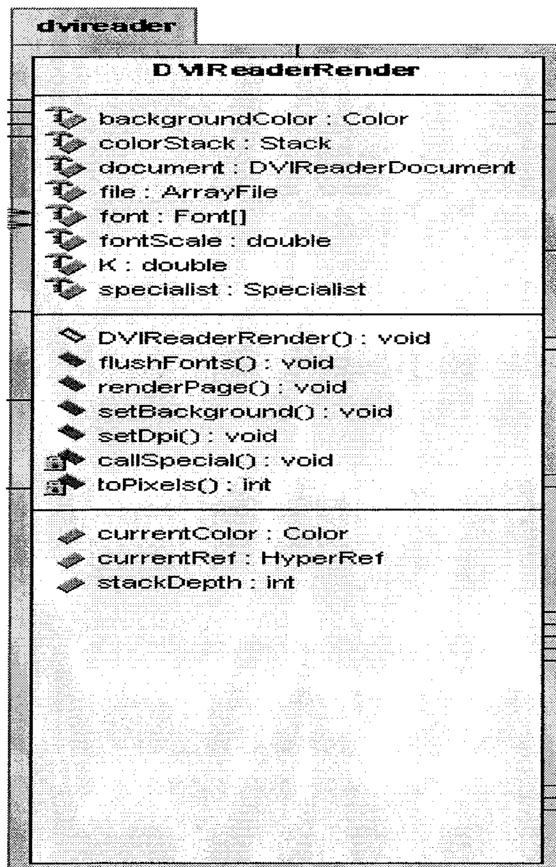

Figure 3.10 ***DVIReaderRender*** **Class**

The ***Bitmap*** **Class** is used to store a bitmap (or a bit array) efficiently. Method *getImage()* is the key function. Figure 3.11 is the UML presentation of the class.

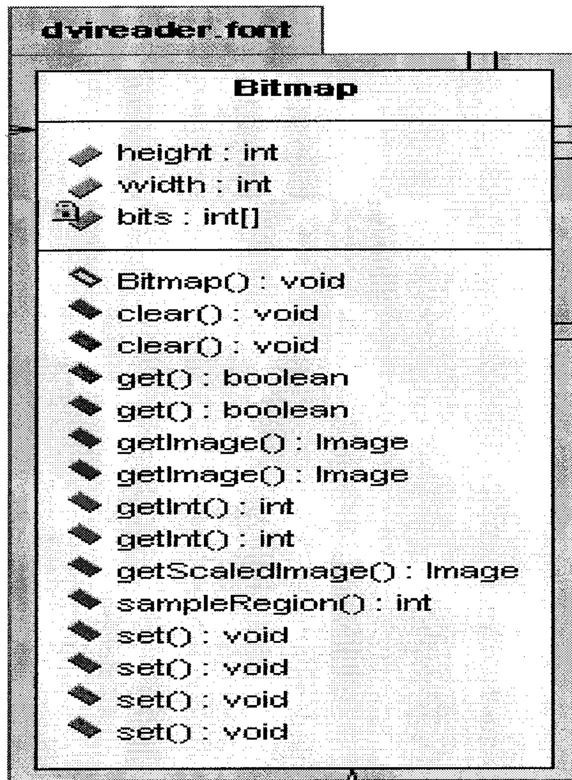

Figure 3.11 ***Bitmap*** **Class**

The ***Font*** **Class** encapsulates the information from a PK font file. Method *loadFont()* knows how to parse a PK file. Figure 3.12 is the UML presentation of the class.

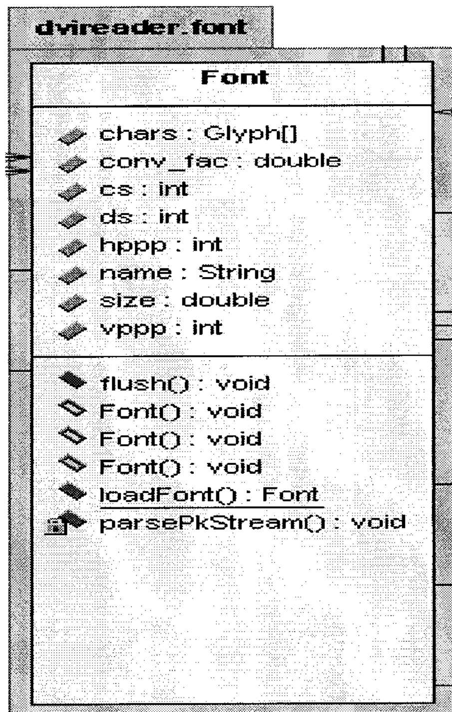

Figure 3.12 ***Font*** **Class**

The ***Glyph*** **Class** represents a single character from a PK font file. Method *readGlyph()* reads individual bits from a *NybbleInputStream*. Figure 3.13 is the UML presentation of the class.

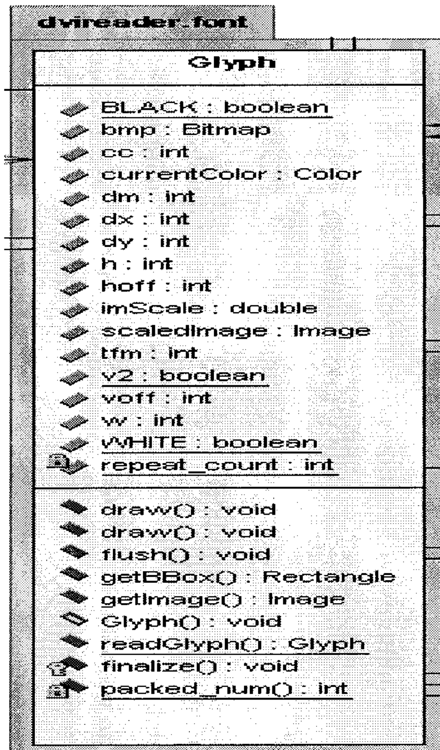

Figure 3.13 ***Glyph*** **Class**

The ***NybbleInputStream*** **Class** encapsulates an input stream and provides methods to read different length values from nybbles to 4 byte ints as needed in parsing PK files. Figure 3.14 is the UML presentation of the class.

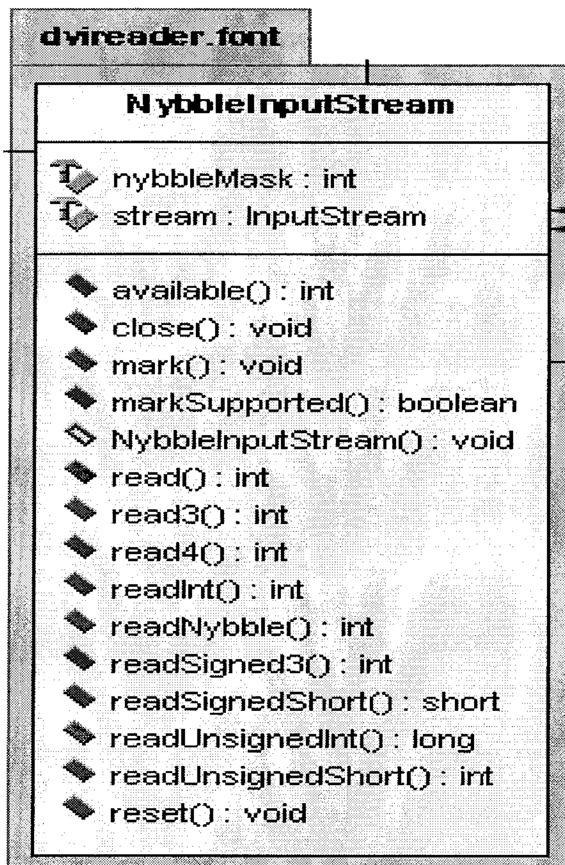

Figure 3.14 ***NybbleInputStream*** **Class**

Key algorithms

loadFont is a method in the Font module, which uses as a given name to return a PK font. Based on the given name, it will initial a *NybbleInputStream* object, which is taken by Method called *parsePkStream* to parse the matched PK fonts to showing back the result. The following are the algorithms of this method.

Algorithm about Font::Font (String s, String s1)

```
1.1 create URL object by parsing input url token s
1.2 for each url token, create URL by parsing the given spec
(which is from input parameter s) within a specified context
(which is generated from step 1.1)
1.2.1 Opens a connection to next URL and returns an inputStream
for reading from that connection.
1.2.2 Returned inputStream was processed by Class
NybbleInputStream.
1.2.3 If object NybbleInputstream is not NULL, then parse object
of NybbleInputStream by using methond Font::parsePkStream, and
return.
```

Algorithm about Font::parsePkStream (NybbleInputStream)

```
2.1 read first byte from the NybbleInputStream
2.2 read second byte from the NybbleInputStream
2.3 if the first byte != 247, or the second byte != 89, then
return with an error message.
2.4 read the 3rd byte from the NybbleInputStream, which is the
length of a block we need to read
2.5 read n bytes (n is from step 2.4)
2.6 read next 4 bytes, and reformat as an int (ds)
2.7 read next 4 bytes, and reformat as an int (cs)
2.8 read next 4 bytes, and reformat as an int (hppp)
2.9 calculate conv_fac as
        conv_fac = ((double)ds / 72057594037927936D) *
(double)hppp;
```

```
2.10 read next byte until the value is 245
2.11 for each byte (j) read from step 2.10 apply the following
logic:
        BEGIN
     if (j < 240) and ((j bitwise AND 3) > 6)
         read 4 byte and reformat as an int
         Glyph.readGlyph(NybbleInputstream,j)

     else if (j < 240) and ((j bitwise AND 3) > 3 and (j bitwise
AND 3) <=6)
         read unsigned short from input stream
         j1 = (j % 4) * 0x10000 + unsigned short
         Glyph glyph1 = Glyph.readGlyph(NybbleInputstream, j);
         chars[glyph1.cc] = glyph1;

     else if (j < 240) and ((j bitwise AND 3) <= 3)
         read one byte from input stream
         int k1 = (j % 4) * 256 + one byte value
         Glyph glyph2 = Glyph.readGlyph(NybbleInputstream, j)
         chars[glyph2.cc] = glyph2

     else if (j=240)
         read one byte from input stream, which is the length of
a block we need to read
         read n (which is from above step) bytes
     else if (j=244)
         read 4 bytes
        END
```

3.3.3 Sequence Diagram in System

The Sequence diagrams are shown in Figure 3.15. They capture the course of events that take place when user selects a DVI file, and even demonstrate most of the activities and transitions that occur during a tool control period. Most of this diagram shows a process to parse and scan the header and body of a DVI file. When an object of *DVIReaderPanel* receives a *initComponnents()* notice from *MainWindow* object, as shown in Figure 3.15, the function *showPage()* will be invoked. Object of *DVIReaderRender* will take the act of *renderPage()*, at the same time the *initDocument()* message will pass to the *DVIReadertDocument* object which will activate *parsePre(), parsePost()* and *scanPage()* to invocate the object.

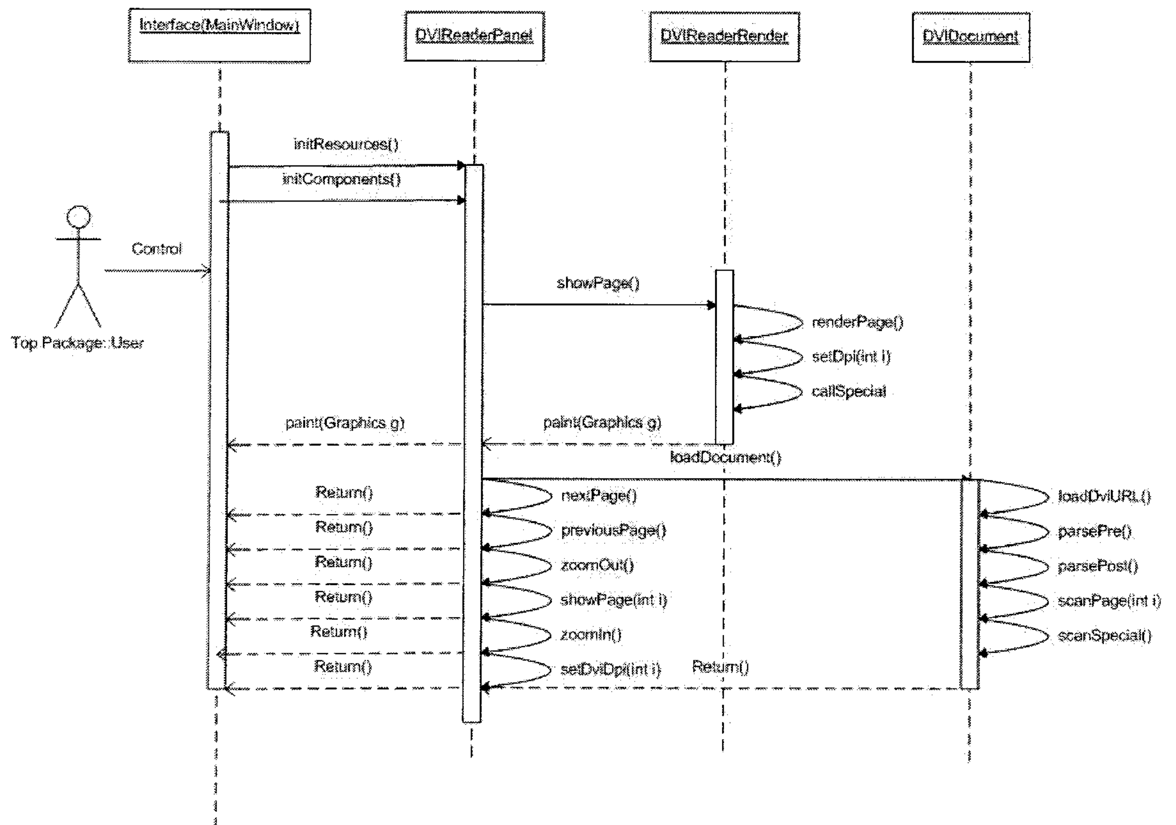

Figure 3.15 Sequence Diagram for System Activities

In Figure 3.16, we describe the sequence of actions that happens when an object of *Font* receives a *parsePost()* notice from *DVIReaderDocument* object. As shown in Figure 3.16, function *loadFont()* and *parsePkStream()* will be invoked. When the message passes to the *NybbleInputStream* object, it will active *read()* and *readGlyph()* to invoke the *Glyph* object. Upon receiving the message, the *Glyph* object will initialize a *Bitmap* object to get a bitmap image.

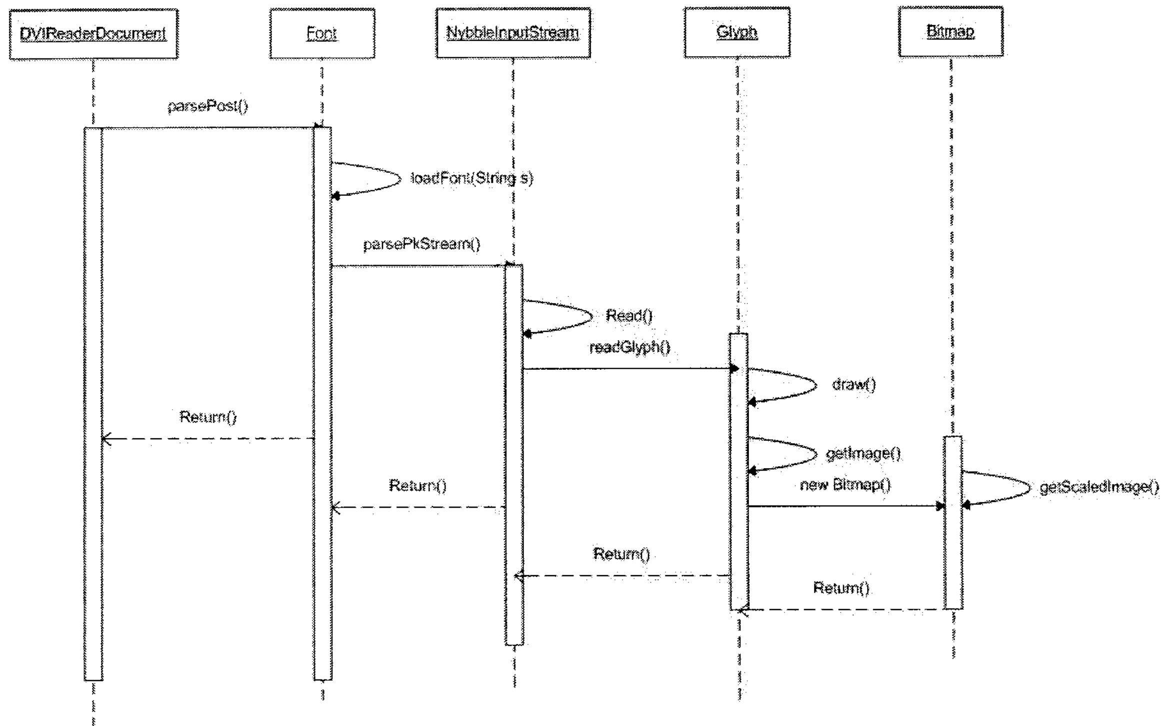

Figure 3.16 Sequence Diagram of Font Module

Chapter 4 Testing Plan and Results

A fully operational *DVIReader* has been implemented based on the modular design discussed in Chapter 3. In this Chapter, we will show how to build a test plan to see how it works. The purposes of testing are to validate and verify this *DVIReader* tool to ensure that the system runs with reliability and efficiency, and the result is reported with accuracy and correctness.

4.1 Environmental Testing

The environment testing is used to determine the hardware and software requirements for this system. The results are listed below:

Hardware:

- CPU: Pentium III Processor or up
- Memory: 256 MB
- Hard DISK Storage: 10 GB

Software:

- Operating System: Microsoft Windows 2000 or XP
- Java Environment: Java(*TM*) 2 SDK, Standard Edition Version 1.4.1

4.2 Functional Testing

Each module or function class of *DVIReader* has been individually tested. In this section, we show the complete system testing with real-world application to ensure the overall robustness and correctness of all aspects of dependability supported by *DVIReader*. Testing was undertaken in Microsoft Windows 2000 and Windows XP; Java Environment: Java(*TM*) 2 SDK, Standard Edition Version 1.4.1.

As a result of this research, we came up with a description of testing *DVIReader* applications. The DVI documents include characters with many parameters and specials may take dozens of variables - text font, color, emphasis, page numbering, name of a template HTML file, scale factor, etc. For testing system communication level, methods *initComponments()* and *initDocument()* have been tested properly. In system testing level, by testing using *parsePre(), parsePost()* and *scanPage()* in the *DVIReadertDocument*, a test of parse head and body of DVI file have been passed properly. Also the same level of testing, act of *renderPage()* for scanning page information of DVI files have been tested properly. Methods *loadFont()* and *parsePkStream()* have got expected result for the font module system testing.

At user interface level, test cases that we used to validate the overall functioning of *DVIReader* are partially listed in Table 4.1.

Test Cases	Initial State	Actions or Events	Expected Results	Pass
Case 1	Window shows	User Open a DVI file	DVI file shows	pass
Case 2	DVI file shows	Go next page	Next page	pass
Case 3	DVI file shows	Go pervious page	Pervious page	pass
Case 4	DVI file shows	Zoom in	Zoom in	pass
Case 5	DVI file shows	Zoom out	Zoom out	pass
Case 6	DVI file shows	Print	Print pages	pass
Case 7	DVI file shows	Set text color	Text's color changes	pass
Case 8	DVI file shows	Go to page	Go to any page	pass
Case 9	DVI file shows	Full		pass
Case 10	DVI file shows	Close File	Close a DVI file	

Table 4.1: Test Cases of *DVIReader*

Chapter 5 Conclusion and Future Work

In this thesis, we have presented *DVIReader*, an object-oriented viewer for (La)TeX 's generic output format known as .dvi, and present the "pages" of the .DVI file one at a time as a slide show on the Windows platform. After we have conducted an investigation of related work, we know that Xdvi and IDVI are not targeted at doing presentations or multimedia. Therefore, the motivation of this thesis design goal is to present a high quality slide show on the Windows platform by loading DVI document. Also, we have presented the background of LaTex; provided documentation for the DVI file format, also we have introduced details of Java implementation for each module according to its sequence diagrams. The *DVIReader* is a portable system, since it is written in Java, and run it within *JAVA Virtual Machine* (JVM) (JVM plays a central role in making Java portable. It provides a layer of abstraction between the compiled Java program and the underlying hardware platform and operating system. The JVM is central to Java's portability because compiled Java programs run on the JVM, independent of whatever may be underneath a particular JVM implementation.), also *DVIReader* is high performance system, for example, we chose Java Patterns for the implementation of *DVIRreader*, in the case of Synchronized Initialization (which we described in Section 1.4) can perform a fast object loading solution in our system. *DVIRreader* can be characterized by the following features: (a) fully object-oriented; (b) modularization; (c) portable; (d) autonomous; (e) high performance.

5.1 Contributions

This work is dedicated to the field of present DVI documents on the application, formatted exactly as they are formatted by TeX. The main contributions are summarized to be:

- Conducted an investigation of related work that helps to identify the problems and to provide a solution targeted at doing presentations or multimedia.
- Designed the specification of functionalities of building DVI format, especially, developed an efficient algorithm for parsing PK font.
- Developed DVI file viewer system by using modular composition.
- Delivered an object-oriented design of *DVIReader* system architecture where each module is independently implemented.
- Delivered an implementation of *DVIReader* in an object-oriented language, is a viewer for (La)TeX 's generic output format known as .dvi, and presents the "pages" of the .dvi file one at a time as a slide show on the Windows platform.
- Collected over 1700 PK fonts from web [8, 9].
- Provide a simple solution to load nearby size font only if *DVIReader* can not find matched PK font.
- Used the System.exec( ) function to start an outside application (put my help file by using System.exec( ) function in the JAR file) [11,12]

5.2 Future Research Directions

Over our experience in designing and developing *DVIReader*, we envision the following research directions:

- Font Generation

It would be nice to have *DVIREADER* invoke font generation machinery if any is available. I've never used a TeX environment with automatic font generation, so I have no idea how useful it is. The DVIReader application needs to know character dimensions, etc. So fonts must be generated before the application runs.

- Support web browse

Since more and more .dvi files are put on the web, by taking advantage of the browser today, it would be good idea to support Applet to view DVI document online.

Bibliography

1. DVI Type: http://www.ctan.org/tex-archive/systems/knuth/texware/dvitype.web, D. E. Knuth last access date: 08/05/2005

2. DVI format: http://www.tug.org/tex-archive/nonfree/dviware/dvi2tty/DVI.format last access date: 08/05/2005

3. DVI Speciation: http://www.math.umd.edu/~asnowden/comp-cont/dvi.html last access date: 08/05/2005

4. TeX-related documentation: http://www.cl.cam.ac.uk/TeXdoc/TeXdocs.html last access date: 08/05/2005.

5. The Comprehensive TeX Archive Network: http://www.ctan.org/, the CTAN team last access date: 08/05/2005.

6. LaTeX Background: http://www.image.ufl.edu/help/latex/intro.shtml, University of Florida College of Liberal Arts & Sciences, last access date: 08/05/2005.

7. xdvi, http://math.berkeley.edu/~vojta/xdvi.html , Paul Vojta, last access date: 08/05/2005.

8. PK Fonts Collections: http://www.math.sc.edu/lib/idvi/pk/, last access date: 08/05/2005.

9. PK Fonts Collections:http://www.tug.org/tex-archive/, last access date: 08/05/2005.

10. S.Mishra, L.L. Peterson, R.D. Schlichting, "Experience with Modularity in Consul", Software Practice and Experience, 1993.

11. Cay S. Horstmann and Gary Cornell Core Java 2, Volume I-Fundamentals, Sun Microsystems, Inc. 2000

12. Cay S. Horstmann and Gary Cornell Core Java 2, Volume II-Advanced features, Sun Microsystems, Inc. 2000

13. *LaTeX User's Guide and Reference Manual* by Leslie Lamport, AddisonWesley Publishing Company, Reading, MA, 2 edition, 1994.

14. *The LaTeX Companion* by Michel Goossens, Frank Mittelbach and Alexander Samarin, Addison Wesley Professional 2004.

15. UML for Java Programmers Robert C. Martin, Pearson Education Inc. 2003.

16. JIT: ***Just-in-time*** *compiler*: http://www.webopedia.com/TERM/J/JIT.html, last access date: 08/05/2005.

17. IDVI, http://www.geom.uiuc.edu/idvi/, Garth A. Dickie, last access date: 08/05/2005.

Appendix a Description of Opcodes [3]

- Opcodes 0-127: set_char_*i* (0 <= i <= 127)

  Typeset character number i from font f such that the reference point of the character is at (h,v). Then increase h by the width of that character. Note that a character may have zero or negative width, so one cannot be sure that h will advance after this command; but h usually does increase.

- Opcodes 128-131: set*i* (1 <= i <= 4); c[*i*]

  Same as set_char_0, except that character number c is typeset. TeX82 uses the set1 command for characters in the range 128 <= c < 256. TeX82 never uses the set2, command which is intended for processors that deal with oriental languages; but DVItype will allow character codes greater than 255, assuming that they all have the same width as the character whose code is c mod 256.

- Opcode 132: set_rule; a[4], b[4]

  Typeset a solid black rectangle of height a and width b, with its bottom left corner at (h,v). Then set h:=h+b. If either a < =0 or b < =0, nothing should be typeset. Note that if b < 0, the value of h will decrease even though nothing else happens. Programs that typeset from DVI files should be careful to make the rules line up carefully with digitized characters, as explained in connection with the rule_pixels subroutine below.

- Opcodes 133-136: put*i* (1 <= i <= 4); c[*i*]

  Typeset character number c from font f such that the reference point of the character is at (h,v). (The put commands are exactly like the set commands, except that they simply put out a character or a rule without moving the reference point afterwards.)

- Opcode 137: put_rule; a[4], b[4]

  Same as set_rule, except that h is not changed.

- Opcode 138: nop

  No operation, do nothing. Any number of nop's may occur between DVI commands, but a nop cannot be inserted between a command and its parameters or between two parameters.

- Opcode 139: bop; c_0[4]..c_9[4], p[4]

  Beginning of a page: Set (h,v,w,x,y,z):=(0,0,0,0,0,0) and set the stack empty. Set the current font f to an undefined value. The ten c_i parameters can be used to identify pages, if a user wants to print only part of a DVI file; TeX82 gives them the values of \count0...\count9 at the time \shipout was invoked for this page. The parameter p points to the previous bop command in the file, where the first bop has p=-1.

- Opcode 140: eop

  End of page: Print what you have read since the previous bop. At this point the stack should be empty. (The DVI-reading programs that drive most output devices will have kept a buffer of the material that appears on the page that has just ended. This material is largely, but not entirely, in order by v coordinate and (for fixed v) by h coordinate; so it usually needs to be sorted into some order that is appropriate for the device in question. DVItype does not do such sorting.)

- Opcode 141: push

  Push the current values of (h,v,w,x,y,z) onto the top of the stack; do not change any of these values. Note that f is not pushed.

- Opcode 142: pop

  Pop the top six values off of the stack and assign them to (h,v,w,x,y,z). The number of pops should never exceed the number of pushes, since it would be highly embarrassing if the stack were empty at the time of a pop command.

- Opcodes 143-146: right*i* (1 <= i <= 4); b[i]

  Set h:=h+b, i.e., move right b units. The parameter is a signed number in two's complement notation; if b < 0, the reference point actually moves left.

- Opcodes 147-151: w*i* (0 <= i <= 4); b[i]

  The w0 command sets h:=h+w; i.e., moves right w units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession. The other w commands set w:=b and h:=h+b. The value of b is a signed quantity in two's complement notation. This command changes the current w spacing and moves right by b.

- Opcodes 152-156: x*i* (0 <= i <= 4); b[i]

  The parameterless x0 command sets h:=h+x; i.e., moves right x units. The x commands are like the w commands except that they involve x instead of w. The other x commands set x:=b and h:=h+b. The value of b is a signed quantity in two's complement notation. This command changes the current x spacing and moves right by b.

- Opcodes 157-160: down*i* (1 <= i <= 4); a[i]

  Set v:=v+a, i.e., move down a units. The parameter is a signed number in two's complement notation; if a < 0, the reference point actually moves up.

- Opcodes 161-165: y*i* (0 <= i <= 4); a[i]

  The y0 command sets v:=v+y; i.e., moves down y units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession. The other y commands set y:=a and v:=v+a. The value of a is a signed quantity in two's complement notation. This command changes the current y spacing and moves down by a.

- Opcodes 166-170: z*i* (0 <= i <= 4); a[i]

  The z0 command sets v:=v+z; i.e., moves down z units. The z commands are like the y commands except that they involve z instead of y. The other z commands set z:=a and v:=v+a. The value of a is a signed quantity in two's complement notation. This command changes the current z spacing and moves down by a.

- Opcodes 171-234: fnt_num_*i* (0 <= i <= 63)

  Set f:=i. Font i must previously have been defined by a fnt_def instruction, as explained below.

- Opcodes 235-238: fnt*i* (1 <= i <= 4); k[i]

  Set f:=k. TeX82 uses the fnt1 command for font numbers in the range 64 < =k < 256. TeX82 never generates the fnt2 command, but large font numbers may prove useful for specifications of color or texture, or they may be used for special fonts that have fixed numbers in some external coding scheme.

- Opcodes 239-242: xxx*i* (1 <= i <= 4); k[i], x[k]

  This command is undefined in general; it functions as a k+i+1$-byte nop unless special DVI-reading programs are being used. TeX82 generates

xxx1 when a short enough \special appears, setting k to the number of bytes being sent. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

- Opcodes 243-246: fnt_def*i* (1 <= i <= 4); k[i], c[4], s[4], d[4], a[1], l[1], n[a+l]

The four-byte value c is the check sum that TeX (or whatever program generated the DVI file) found in the TFM file for this font; c should match the check sum of the font found by programs that read this DVI file.

Parameter s contains a fixed-point scale factor that is applied to the character widths in font k; font dimensions in PK files and other font files are relative to this quantity, which is always positive and less than 2^27. It is given in the same units as the other dimensions of the DVI file. Parameter d is similar to s; it is the "design size", and (like s) it is given in DVI units. Thus, font k is to be used at mag s / 1000 d times its normal size.

The remaining part of a font definition gives the external name of the font, which is an ASCII string of length a+l. The number a is the length of the "area" or directory, and l is the length of the font name itself; the standard local system font area is supposed to be used when a=0. The n field contains the area in its first a bytes.

Font definitions must appear before the first use of a particular font number. Once font k is defined, it must not be defined again; however, we shall see below that font definitions appear in the postamble as well as in the pages, so in this sense each font number is defined exactly twice, if at all. Like nop commands, font definitions can appear before the first bop, or between an eop and a bop.

- Opcodes 247: pre; i[1], num[4], den[4], mag[4], k[1], x[k]

  The preamble contains basic information about the file as a whole and must come at the very beginning of the file. The i byte identifies DVI format; currently this byte is always set to 2. (The value i=3 is currently used for an extended format that allows a mixture of right-to-left and left-to-right typesetting. Some day we will set i=4, when DVI format makes another incompatible change - perhaps in the year 2048.)

  The next two parameters, num and den, are positive integers that define the units of measurement; they are the numerator and denominator of a fraction by which all dimensions in the DVI file could be multiplied in order to get lengths in units of 10^(-7) meters. (For example, there are exactly 7227 TeX points in 254 centimeters, and TeX82 works with scaled points where there are 2^16 sp in a point, so TeX82 sets num=25400000 and den=7227 2^16=473628672.

  The mag parameter is what TeX82 calls \mag, i.e., 1000 times the desired magnification. The actual fraction by which dimensions are multiplied is therefore m n /1000 d. Note that if a TeX source document does not call for any true dimensions, and if you change it only by specifying a different \mag setting, the DVI file that TeX creates will be completely unchanged except for the value of mag in the preamble and postamble. (Fancy DVI-reading programs allow users to override the mag setting when a DVI file is being printed.)

  Finally, k and x allow the DVI writer to include a comment, which is not interpreted further. The length of comment x is k, where 0 < = k < 256.

- Opcodes 248: post; p[4], num[4], den[4], mag[4], l[4], u[4], s[2], t[2]; < *font definitions* >

The last page in a DVI file is followed by post; this command introduces the postamble, which summarizes important facts that TeX has accumulated about the file, making it possible to print subsets of the data with reasonable efficiency. The parameter p is a pointer to the final bop in the file. The next three parameters, num, den, and mag, are duplicates of the quantities that appeared in the preamble.

Parameters l and u give respectively the height-plus-depth of the tallest page and the width of the widest page, in the same units as other dimensions of the file. These numbers might be used by a DVI-reading program to position individual "pages" on large sheets of film or paper; however, the standard convention for output on normal size paper is to position each page so that the upper left-hand corner is exactly one inch from the left and the top. Experience has shown that it is unwise to design DVI-to-printer software that attempts cleverly to center the output; a fixed position of the upper left corner is easiest for users to understand and to work with. Therefore l and u are often ignored.

Parameter s is the maximum stack depth (i.e., the largest excess of push commands over pop commands) needed to process this file. Then comes t, the total number of pages (bop commands) present.

The postamble continues with font definitions, which are any number of fnt_def commands as described above, possibly interspersed with nop commands. Each font number that is used in the DVI file must be defined exactly twice: Once before it is first selected by a fnt command, and once in the postamble.

- Opcodes 249: post_post; q[4], i[1]; 223's

The last part of the postamble, following the post_post byte that signifies the end of the font definitions, contains q a pointer to the post

command that started the postamble. An identification byte, i, comes next; this currently equals 2, as in the preamble.

The i byte is followed by four or more bytes that are all equal to the decimal number 223 (i.e., 337 in octal). TeX puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a DVI file makes it feasible for DVI-reading programs to find the postamble first, on most computers, even though TeX wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the DVI reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read q, and move to byte q of the file. This byte should, of course, contain the value 248 (post); now the postamble can be read, so the DVI reader discovers all the information needed for typesetting the pages. Note that it is also possible to skip through the DVI file at reasonably high speed to locate a particular page, if that proves desirable. This saves a lot of time, since DVI files used in production jobs tend to be large.

Appendix B User Manual and Installation

- DVIReader can be started directly from the jar archive via " java -jar DVIReader.jar " in *DOS* line

After tool starts up, the window of main frame will pop up. The running procedure is as below:

- Click on part of the document to see a magnified view in the ***DVIReader*** Controls window.
- Click and drag page in the ***DVIReader*** Controls window to adjust your position.
- The buttons at the top of the **Button Controls** window allow you to change the scale factor (zoom in and out), or go to another page. The buttons labeled - and + change the scale factor. The buttons labeled <-, ->, jump to the previous page, the next page, and the last page, respectively. To go to a specific page, type the page number into the text field and press enter.
- In addition to using the buttons in the ***DVIReader*** Controls window, you may navigate using a subset of the keyboard shortcuts available in ***DVIReader***. A shortcut description table is listed in Table A:

Shortcut	Operation Taken
Ctrl O	Open file
Ctrl W	Close file
Ctrl P	Print
Ctrl I	ZoomIn
Ctrl U	ZoomOut
Ctrl Z	Go to the first page.
Ctrl Y	Go to the last page.
Ctrl F	Go to the page N.

Table A: Shortcuts Descriptions of *DVIReader*

- Same functions are used in menu bar; also you can set color of DVI file text by select *Menu Format.*

- **Installation**

  You can start up with following DOS line commands:

% java -classpath /somedir/DVIReader.jar

%java -jar DVIReader.jar

However you might want to add the jar file to your {CLASSPATH} environment

- For PK font package, you need to unzip PKFonts.zip to your directory. By default, we set this directory to E:\pkfont, if it is necessary, you can change it by modifying *MainWindowResource.properties* under \dvireader\resources\bundles\ in the *DVIReader.JAR* file make change in the line *"dvireader.font.path=file:/E:/pkfont"* to your directory and put it back to the Jar file.