The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Working Paper 240 March, 1983

An Empirical Study of Program Modification Histories

Linda M. Zelinka

ABSTRACT

Large programs undergo many changes before they run in a satisfactory manner. For many large programs, modification histories are kept which record every change that is made to the program. By studying these records, patterns of program evolution can be identified. This paper describes a taxonomy of types of changes which was developed by studying several such histories. In addition, it discusses a possible application of this classification in an interactive tool for the updating of user documentation.

A.I. Laboratory Working Papers are produced for internal circulation, and may contain information that is, for example, too preliminary or too detailed for formal publication. It is not intended that they should be papers to which reference may be made in the literature.

Acknowledgments

I would like to thank Chuck Rich for giving this project direction by generously sharing his time and providing key insights. Likewise, the helpful suggestions made by Dick Waters and Chuck Rich who proofread this paper were very much appreciated.

The opportunity to conduct this research was provided by the MIT UROP Program.

CONTENTS

1. The Taxonomy 3

1.1 Example of a Modification History 3
1.2 Additive/Nonadditive 5
1.3 Level of Description Problem 5
1.4 Levels of Description 6
1.5 Where-What-Why-How 7

2. Applications 11

2.1 Updating User Documentation 11
2.2 Scenario 14

Introduction

Large programs usually undergo many changes before they run in a satisfactory manner. For many large programs, modification histories are kept which record every change that is made to the program. By studying these records, patterns of program evolution can be identified. This paper will describe a taxonomy of types of changes which was developed by studying several such histories. In addition, it will discuss a possible application of this classification in an interactive tool for the updating of user documentation.

This paper focuses on how large programs are modified by expert programmers. This is part of a larger study of program structures which is being conducted by the Programmer's Apprentice group in an attempt to develop a computer assistant for programmers. The goal of the group is to understand not only how programs are typically modified, but also how they are analyzed, synthesized, explained, verified, and documented.

Since the source code for the programs in this study was not accessible, the programs were dealt with indirectly through modification histories and also through user documentation. A level of abstraction was maintained by not becoming entangled in the details of how the change was actually implemented. In addition, the categories of changes were kept general by using this approach.

1. The Taxonomy

1.1 Example of a Modification History

Figure 1 shows a portion of a typical modification history. It is from a program called Atsign (@).

The Atsign program makes cross-reference listings of programs. Given a file or several files, it will print on each line of the file(s), the location (file, page, and line) of the definition of one of the symbols which is referred to on that line. Atsign attempts heuristically to choose the most "interesting" symbol to cross reference. A title page, subtitle table of contents, a symbol table, and a cross reference table are included in the listing. If the files which compose a program are modified after an initial listing of the program has been made, then an incremental listing can be made which will include page maps. These list the numbers of all pages printed, and separately list the numbers of all pages created or deleted.

The Atsign program takes a command string which is made up of a list of switches. Several switches may be enclosed in parentheses or else each switch is prefixed with a "/", followed by a single character. Some switches take arguments which are placed either just before the single character (e.g. /1G) or else in [ ]'s immediately following the single character (as in /D[Dover]).

When making a listing, the user must specify the names of the files to be listed, the language they are written in, and various other parameters. All this information is remembered in a special "LREC" file just for that purpose. A typical invocation of Atsign is:

```
:@ SPZ;FOO LREC/G/L[MUDDLE]/F[20FG],SPZ;FOO>,BAR>,BLETCH>
```

"SPZ;FOO LREC/G" says that the LREC file should be called SPZ;FOO LREC. The rest of the command string says that the program to be listed is contained in files SPZ;FOO, BAR, and BLETCH, is written in MUDDLE, and should be printed in font 20FG.

Referring to Figure 1, the modification history includes for each modification: the date the modification was made, who made it, and a description of the change. This particular history covers changes made to Atsign over a period of nearly six years from March, 1976 to December, 1981.

Another modification history which was examined closely in this study is that of the BABYL (pronounced

```
SUBTTL  BUREAUCRACY: WHO DID WHAT TO @ WHEN

;;; ***** PEOPLE WHO HAVE HACKED THE PROGRAM *****
;;; GLS         Guy L. Steele Jr.    (GLS@MIT-MC)
;;; RMS         Richard M. Stallman (RMS@MIT-AI)
;;; RHG/RG02    Richard H. Gumpertz (Gumpertz@CMU-10A)
;;; MRC         Mark Crispin        (MRC@SU-AI)
;;; MOON        David A. Moon       (MOON@MIT-MC)
;;; EAK         Earl A. Killian     (EAK@MIT-MC)
;;; MT          Michael Travers      (MT@MIT-XX)
;;; JMN         Joseph M. Newcomer  (Newcomer@CMU-10A)
;;; KLH         Ken Harrenstien      (KLH@MIT-AI/SRI-NIC)

;;; THE AUTHORITATIVE SOURCE FOR @ IS [MIT-AI]QUUX;@ >

;; WARNING: RMS, MRC, AND GLS DON'T TAKE THIS BUREAUCRACY VERY SERIOUSLY.

;;; ***** Modification History *****
;;;   Date       Who      Description
;;;    -          -       Modifications prior to 28 Mar 76 went unrecorded
;;; 28 Mar 76    RHG      Redid line number checking
;;;    "          "       Fixed bug in /-T caused by line number hacking
;;;    "          "       Added PDL overflow handling for BOTS
;;;    "          "       Added "extended LOOKUP" code under BOTS
;;;    "          "       Added creation date printing to PTLAB for BOTS
;;; 29 Mar 76     "       Added DROPTHRUTO macro
;;; 30 Mar 76    RMS      Clean up problems in ITS version introduced by above.
;;; 01 Apr 76    RMS      Added /L[PL/I]
;;; 01 Apr 76    RMS      Displays info on progress of listing in the .WHO variables.
;;;    "          "       /nS sets symbol space to <n> symbols.
;;; 03 Apr 76     "       PTLAB made more subroutinized, and more uniform across versions.
;;;    "          "       1st line of continuation pages is never used for text.
;;;    "          "       Date appears on sym tab, CREF, SUBTTL table of contents, ...
;;;    "          "       Infamous excess almost-blank page bug fixed.
;;; 06 Apr 76    RHG      Added /K switch support, redid CKLNM (again -- sigh)
;;;    "          "       Suppressed checksumming of line numbers, except under /K switch
;;; 07 Apr 76     "       Fixed bug in last changes to checksumming, CKLNM
;;;    "          "       Simplified PTLO hacking for TWOSEG
;;;    "          "       Fixed date setting for BOTS copyrights
;;;    "          "       Added SITNAM stuff
;;;    "          "       Fixed /nS printout on title page
;;;    "          "       Fixed bug causing last page to always be printed under BOTS
;;; 26 Apr 76    MRC      Fixed PPN printout lossage under BOTS
;;; 15 Jun 76    Moon     Added /L[UCONS]
;;; 05 Sep 76    MRC      Fixed assembly error in BOTS
;;; 06 Sep 76    RMS      OBARRAY assembled without literals
;;;    "          "       LISPSW conditional to save space in DEC version
;;; 07 Sep 76     "       SAIL PPN's, font files and XGP commands
;;;    "          "       /X[QUEUE]
;;; 19 Sep 76    MRC      Fixed SAIL PPN's, and pretty cases
;;;                       Installed(and debugged) RMS' written in patches
;;; 02 Oct 76    RMS      Made SAIL version work.  Understand ETV directories & padding.
;;;                       /L[TEXT]
;;; 18 Oct 76    RHG      Made PGNSPC include space for PPN, in CMU version
;;;              RMS      Made automatic queueing work in SAIL version
;;;                       Understand that a narrow font 0 means more room for text
;;;                        (But doesn't work yet - see comment in FNTCPT)
;;;                       On DEC system, "FOO" specifies either null extension or default.
;;;                       Except on ITS, don't use top line of page for text.
;;; 24 Dec 76    RMS      /Y means always print real page #, not virtual.
;;;                       Output file names don't default stickily;  defaulted at
;;;                       open time directly to the /O[...] names.
```

Fig. 1. First Page of Atsign's Modification History

"babble") electronic mail system. BABYL allows a user to read and send mail. In addition, the user may save or delete messages, edit their text, or categorize them by attaching labels. Every message is made up of a "header" part and a "text" part. The header consists of the date, sender, recipient, and subject. The text is the body of the message.

This study analyzed the modifications made to BABYL from the end of February, 1979, up to the middle of March, 1982.

1.2 Additive/Nonadditive

A first approach to categorizing changes made to programs is to classify them as being either *additive* or *non-additive*. An additive change causes the program to do something extra which it didn't do before. The key idea is that after an additive change is made, the program continues performing all the tasks it did prior to the change. Here is an example (all the following examples are from Atsign until mentioned otherwise):

"Added /L[TEXT]."

This modification gives the /L[<language>] switch the ability to handle an input file which is the output from a text justifier. Atsign is still able to list files written in the languages it recognized before the additive modification was made. However, the new and improved Atsign program can also list text justifier output.

In a non-additive change, on the other hand, some most likely unwanted behavior is lost. For example:

"Infamous excess almost-blank page bug fixed."

Modifications which fix bugs are usually non-additive.

1.3 Level of Description Problem

The additive/non-additive categorization immediately runs into problems because in reality modifications which are either purely additive or purely non-additive are quite rare. Most changes have both qualities, such as this one:

"Made /nA print symbol table truncating symbols to n characters."

This change is additive in one sense, because the /A switch has been made able to take an argument. The /nA switch will be able to do more than the /A switch.

A symbol table is an alphabetical table of all symbols showing where (and how) they were defined. It is printed with every listing unless the user specifies that it should be suppressed. Frequently, a few long symbol names cause the symbol table to be printed using wide columns. Because of this, only a few columns are able to fit on each page and many pages are needed. Here, an additive improvement was made to the /A switch so that the new /nA switch can truncate symbols before they are printed in the symbol table. This allows space for more columns on each page and thus fewer pages are needed.

In another sense, this modification is non-additive since certain characters in long symbol names will no longer be printed. The /A switch causes Atsign to print every character in the symbol names. The /nA switch, on the other hand, suppresses the printing of more than *n* characters per symbol name. When /nA is used instead of /A, a portion of output behavior is lost.

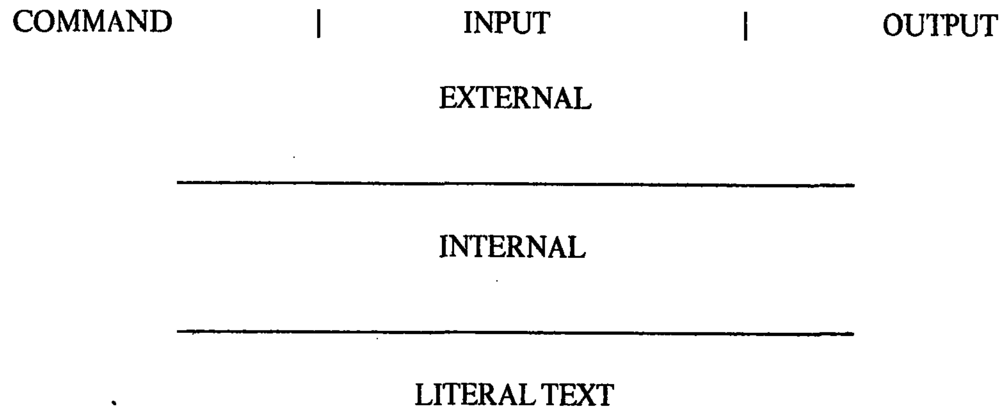

Fig. 2. Levels of Description

1.4 Levels of Description

The simultaneous additive/non-additive nature of modifications makes it necessary for their analysis to take place on different levels of description. Changes that are additive at one level may be non-additive on another. This study found three useful levels. These are listed in Figure 2.

The highest is the *external* level. It is the highest of the three levels in the sense that changes occurring on this level are the most visible to the user. The varying degrees of user visibility which characterize changes will be discussed in more detail in a later section.

Within the external level, changes are placed in three categories: *command*, *input*, and *output*. Changes falling into the first of these include adding or improving a command. For instance,

> "Added /M[<left>,<right>,<top>,<bottom>] to set the margins."

The switch /M[...] is added to specify the page margin sizes.

The next category involves changes to *input*. Modifications in the handling of files or data being accessed by the system fall into this category. For example, the following change specifies how to handle a certain type of input file:

> ".LIBFIL in an assembler-language file means ignore the file completely, if it isn't being listed."

Modifications in the third category deal with *output*. These include changes in the way output is formatted and in what is to be output. For example,

> "Changed date printing format to not use abbreviations."

The next level of description below external is the *internal* organization level. This level deals with, among other things, the order in which subroutines are called, the control of data flow, and the execution of algorithms. For instance,

> "Made (the subroutine) SYN take args in right order (old, new)."

The lowest level involves the *literal text* of the program. Changes on this level include typographical corrections and the rearrangement of code. Most changes come down to being on this level at some point, but only those changes which have no effect on other levels were classified as being on the literal text level, for example,

> "Fixed a typo that caused /Y to turn on magically."

1.5 Where-What-Why-How

Levels of description tell *where* a change takes place. This suggests other questions:

-- *What* does the modification do?
-- *Why* is it made?
-- *How* does it affect the user?

These questions are helpful in finding further classifications of changes.

1.5.1 What

In answer to the question "What does a change typically do?", four kinds of answers can be provided. (See Figure 3.) For one, a change may *add* a feature. For example,

> "Changed /M[...] to have a fifth margin -- the "hole" margin. It is added to either the LEFT or TOP margin as appropriate."

A change may also *merge* things, such as subroutines or commands:

> "Modified /Q switch such that /0Q is the same as /Q."

(/Q causes a copyright message to be printed at the bottom of each page.)
Thirdly, a change may *delete* an unwanted feature. For example,

> "Suppressed the blank page which was printed if /Z but no Table of Contents to print."

(The /Z switch tells Atsign to print a table of contents.)
Finally, a modification may *rearrange* something. The change is able to do this in three different ways.

1) It may *rename*:

> "↑S K changed to ↑S L."

(In the BABYL mail system, the ↑S L command reads a label which is attached to a message.)

2) A modification may *reorder*, just as this one does:

> "Moved some definitions around."

3) It can also *reformat* something, most often output or text:

"Month and day names abbreviated to fit in field on the Dover (printing device)."

ADD FEATURE

MERGE

DELETE

REARRANGE

-- Rename

-- Reorder

-- Reformat

Fig. 3. What Changes Do

Adding, merging, deleting, and rearranging are what changes commonly do on all levels of description. Now, why are they made?

1.5.2 Why

There are many possible reasons for making changes to a program. Some of these reasons are listed in Figure 4. One motive is to *improve* the system's performance. Here's an example from BABYL's modification history:

"If there's no ↑_ at end-of-file BABYL will still warn but will insert it if continued."

BABYL now behaves much better when it finds no control character (↑_) signalling the end of the file.

Improve is a very general reason and can be refined further. To be more specific, the improvement may, among other things, increase *efficiency*,

"E should be twice as fast since corrects message# variables without another pass over file."

In BABYL, the E command expunges deleted messages. After the change was made, E was able to perform faster since it could update certain variables without going through the mail file again.

The improvement may also make something more *convenient*, as in this thoughtful revision:

"Changed (the subroutine) DFLMAR to 1 inch to allow for hole punching."

Thirdly, it may improve by giving *extra functionality*:

"Changed (the subroutine) DATOUT to also print a time."

IMPROVE

More efficient

More convenient

Extra functionality

FIX-BUG

Make promised feature work

Fix unanticipated problem

CLEANUP

CONFIGURE

Fig. 4. Why Changes Are Made

The majority of changes examined in this study were made in order to improve the program. Another common reason was to *fix* a bug. For example, in Atsign, a problem in the 1SUBTO subroutine was corrected:

"Fixed 1SUBTO to skip spaces, not everything else."

Like *improve*, *fix-bug* is a very general reason. In particular, fixing a bug may involve:

1) making a promised feature work,

"Made /M[...] switch actually do something."

(Apparently, the /M switch was failing to provide the expected margins.)

2) or fixing an unanticipated problem.

"Fixed a bug I introduced accidentally in (the subroutine) ENDUND."

A third typical reason for making a change is to *cleanup* the program. For instance, in the BABYL program:

"Flushed superfluous Reply-to: and Sender: headers."

Another motive which a person who is modifying a program may have is to *configure* the program for different sites, machines, or peripheral devices. For example,

"Made <-- an alias for _ so that backarrow and underscore will both win at SAIL."

(SAIL is an acronym for Stanford Artificial Intelligence Laboratory.)

1.5.3 How

Besides classifying changes according to the motives of the programmer, it is useful to describe modifications with respect to *how* they affect the user. An important distinction to be made is between changes whose effects are *visible* to the user and those that are not. (See Figure 5.) Changes made on the Internal and Literal Text levels are usually not user-visible. For example,

"Got rid of duplicate definition of (the function) PTQDAT."

This change will not be noticed by the user. It is only made to cleanup the code, which is of no interest to the user.

An example of a user-visible change is this one:

"Changed CMU's prompt back to '@'."

Observant users of Carnegie Mellon University's version of Atsign might notice the different prompt. It is important to realize, however, that although the effect of this change is user-visible, the user probably won't care about it. The fact that the prompt is an Atsign is probably not interesting to the user or relevant to what he needs to know to use the Atsign program successfully.

Consider the example:

"/X now means 'treat as a graphics device and default to XGP'. It takes no
other args."

The /X switch means the Atsign program is to use the graphics features of the output device, and the default device is the XGP if not otherwise specified. This modification occurred about a month and a half after a change took place which made /0X indicate that the Dover (another graphic device) was to be used as the output device. Since the meaning of the /X switch was changed by this modification, the users of Atsign had to be notified of the change. User documentation, which explained how to use the Atsign program, had to be updated. Most likely, messages were sent out to all users in order to alert them about the change. This example makes it clear that there is a certain subclass of user-visible changes which are important for the user to know about.

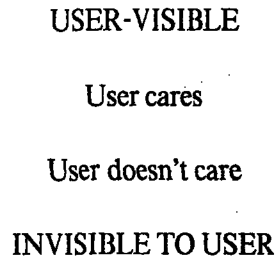

Fig. 5. How Changes Affect the User

2. Applications

Now that some classifications of answers have been provided for the questions *where*, *what*, *why*, and *how*, the next question to consider is what to do with this information. Changes can be placed in certain categories dealing with function, motivation, and effect on several different levels of description. How can this taxonomy be used?

2.1 Updating User Documentation

One possible application of the vocabulary of changes is in the maintenance of up-to-date user documentation. Figure 6 shows the beginning of Atsign's user documentation. It is in a standardized, structured format used by the MIT AI Lab's INFO program. This is an information system which gives the user access to a large tree of documentation on the various programs available to him.

Atsign's documentation explains everything the user needs to know in order to make a cross-reference listing of a program. It is made up of several sections which provide information on various topics. These topics are listed in the menu at the beginning of the section.

The user relies on accurate documentation in order to be able to use the program successfully. The problem is that this documentation gets obsolete quickly because of the high rate at which modifications are made.

After making a change, it would be nice for a programmer to be able to immediately update the relevant part of the user documentation without having to manually search through the entire documentation each time to find what needs to be rewritten or added.

In general, portions of a program's code correspond to the sections of user documentation which describe them. (See Figure 7.) This is a many-to-many mapping, i.e., a piece of code may be explained by a section or several sections of user documentation. At the same time, one section of documentation may correspond to many pieces of code.

Changes recorded in modification histories are also associated with pieces of code. A modification logically points to the piece of code which was modified or added. If this section of code is described by a portion of user documentation, then that portion should be shown to the person doing the modifying so that it can be updated.

```
;;; -*-Mode:Lisp; Package:User; Base:10; Fonts:MEDFNB; -*-
^_
File: @  Node: Top,  Up: (DIR),  Next: Basic

The @ program makes cross-reference listings of programs.  One or many
files can be listed together.  On each line will appear the file, page,
and line of the definition of a symbol referenced on that line.
Comparison listings can be made, containing only the pages changed since
the previous listing, for both programs and text-justifier output files.

* Menu:

* Basic::     Simplest usage of @, for programs.
* Output::    What @ output looks like, for programs.
* Text::      Simplest usage of @, for papers.
* Assembler::What @ understands about assembler language.
* Lisp::      What @ understands about Lisp.
* Muddle::    What @ understands about Muddle.
* Quotes::    Quotes (') control whether files are listed.
* Switches:: What you can do with command line switches.
* Files::     What files are used, and their default names.
* Substitutions::  Adding, removing or changing names of source files.
^_
File: @  Node: Basic,  Up: Top,  Previous: Top,  Next: Output

Fundamentals of @ for programs:

When you make a listing, you must initially specify the names of the
files to be listed, the language they are written in, and various other
parameters.  All of this information, is remembered in a special "LREC"
file just for that purpose.  To make an update listing, you need only
specify the name of the LREC file;  all other information is read from
there.

Comparison listings work by keeping in the LREC file a table of
checksums of all the pages of each source file.  When it is time to make
the update listing, the new source file's pages' checksums are compared
with those listed in the LREC file.  A page whose checksum has changed
needs to be listed.  This scheme means that it is not necessary to
retain the copy of the program which was last listed.  The LREC file is
all that is necessary for making an update listing.

                          .
                          .
                          .
```

Fig. 6. Beginning of Atsign's User Documentation

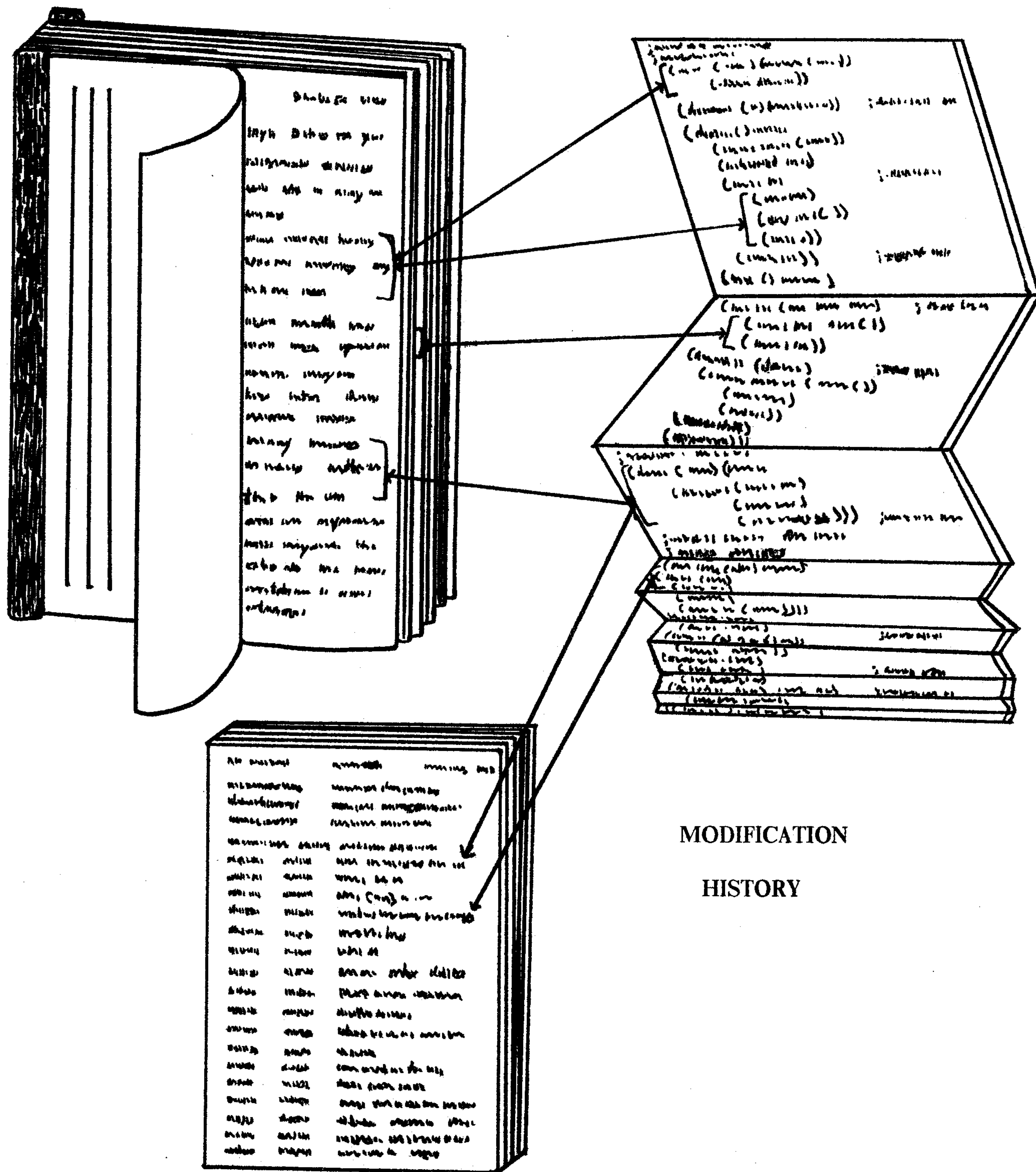

Figure 7.

2.2 Scenario

A proposed tool for making the updating of user documentation more efficient and complete categorizes the documentation in a way similar to the changes, according to the level of description on which they are analyzed. The structured format of the documentation lends itself toward this type of categorization. For example, the section on switches is labeled as being on the command level. Each paragraph in the section is likewise classified. When a change is made, it would then be possible to zoom in on relevant sections of the documentation and present them to the programmer for editing.

An example scenario will illustrate the intended operation of this tool. Suppose a programmer makes a modification to the Atsign program. The system first asks for a description of the change (system printing in upper case, user in mixed case):

DESCRIBE MODIFICATION:
>*Made /nA print symbol table, truncating symbols to n characters.*

The programmer is then be asked a series of questions followed by a menu of possible answers. The programmer selects the correct answer. In this scenario, a pointing device, called a mouse, is used. The mouse controls a cursor (↑) on the screen. When the cursor points to an item in the menu, a box is drawn around it. The item can then be selected by clicking one of the buttons on the mouse.

The next question the programmer is asked is whether or not the change is user-visible, since only changes which are visible to the user necessitate updating user documentation.

IS THE CHANGE VISIBLE TO THE USER? [Y] N

The user will certainly see the shortened symbol names in the symbol table.

The system also inquires if the user cares about the change. A piece of information may be documented, but it may be such a trivial detail that the user would only be irritated by announcements of its changing.

WILL THE USER CARE ABOUT THE CHANGE? [Y] N

The new /nA switch is important for the user to be told about. Announcements should be sent out immediately to inform users of the change.

Since this change definitely has an effect on the user, the system can be sure that an update of user documentation is called for. It begins by asking *where*, *what*, and *why* questions to obtain more information about the modification.

First, the level of description is asked for. Since the change may have taken place on more than one level, the menu is a multiple one which allows the user to select as many items as he needs. The system will deal with each of those selected one at a time.

ON WHICH LEVEL OF DESCRIPTION DID THE CHANGE TAKE PLACE?

[External] Internal Literal Text

Most of the time, changes which affect the user documentation occur on the external level. Because this level can be further classified, the system is able to ask for a specific category into which the modification falls. Again, the menu is a multiple one, since the change may be placed in more than one category.

```
-*-Text-*-
File: @   Node: Top,   Up: (DIR),   Next: Basic

The @ program makes cross-reference listings of programs.  One or many
files can be listed together.  On each line will appear the file, page,
and line of the definition of a symbol referenced on that line.

* Menu:

* Basic::          Simplest usage of @, for programs.
* Output::         What @ output looks like, for programs.
* Text::           Simplest usage of @, for papers.
* Lisp::           What @ understands about Lisp.
* Muddle::         What @ understands about Muddle.
* Switches::       What you can do with command line switches.
* Files::          What files are used, and their default names.
* Substitutions:: Adding, removing or changing names of source files.
v
File: @  Node: Basic,  Up: Top,  Previous: Top,  Next: Output

Fundamentals of @ for programs:

When you make a listing, you must initially specify the names of the
files to be listed, the language they are written in, and various other
parameters.  All of this information, is remembered in a special "LREC"

                              .
                              .
                              .

File: @  Node: Switches,  Previous: Quotes,  Up: Top,  Next: Files
Table of Switches:

(* indicates a per-file switch; all others are global switches; default
is off (-X) unless otherwise stated):

 /!      Controls handling of input files found not to exist.
         It says to forget all about the file if the user
         doesn't supply a new name (that is, if he types just
         CR) when the error message asks for one.
 /-!     means don't bother complaining; don't list the
         file but do remember it.
                              .
                              .
                              .

 /A      Arbitrarily long symbols should be kept by @; this is
         the default for LISP code.
 /<N>A   Arbitrarily long symbol names should be retained by @,
         but when the symbol table is printed symbol names
         should be truncated to <N> characters, so as to fit
         more columns and use up fewer pages.
 /-A       should remember only the first 6 characters of
         symbol names.  This is the default for MIDAS code.
                              .
                              .
                              .
```

Fig. 8. Portion of Atsign's Documentation: Switches

WHERE ON THE EXTERNAL LEVEL?

[Command] Input **[Output]**

The system deals first with the command category. It will later inquire about the other selected categories until all false pieces of information are ferrcted out of the documentation.

REGARDING COMMANDS:
WHAT DID THE CHANGE DO?

[Add feature]	**Delete**	**Merge**
Reformat	**Rename**	**Reorder**

WHY WAS THE CHANGE MADE?

Cleanup **Configure** **Fixbug** **[Improve]**

SPECIFY: Convenient **Efficient** **[Extra functionality]**

By adding a feature to the /A switch, a new switch, /nA, has been created. The /A switch allows Atsign to keep arbitrarily long symbol names. The improved /nA switch does what /A does, but in addition it truncates the symbol names when the symbol table is printed.

Once the system knows these facts about the change, it can find the section of Atsign's documentation dealing with commands. This section has a list of paragraphs explaining each switch in alphabetical order. The system asks the name of the switch so that the part of the list where /A is explained can be highlighted. The programmer can then insert the description of the /nA switch into the list. (See Figure 8.)

Next, the system turns to the second selected category -- output -- and again asks *where*, *what*, and *why* questions.

REGARDING OUTPUT:
WHAT DID THE CHANGE DO?

Add feature	**[Delete]**	**Merge**
Reformat	**Rename**	**Reorder**

WHY WAS THE CHANGE MADE?

[Cleanup] **Configure** **Fixbug** **Improve**

The system now knows that a deletion was made to clean up the output of the program. Several different features are described in the section of user documentation concerned with output. These are listed at the beginning of the section. (See Figure 9.) Since the documentation has been broken down into these categories, the system can ask for a more specific description of where the change took place within the output category.

WHERE IN OUTPUT?

Title Page	**Page Maps**	**Subtitle Table of Contents**
Listed Source	**[Symbol Table]**	**Title Page**

It is discovered that the symbol table is the output feature on the output level which was cleaned up by a

```
-*-Text-*-
v
File: @  Node: Top,  Up: (DIR),  Next: Basic

The @ program makes cross-reference listings of programs.  One or many
files can be listed together.  On each line will appear the file, page,
and line of the definition of a symbol referenced on that line.

* Menu:
* Basic::        Simplest usage of @, for programs.
* Output::       What @ output looks like, for programs.
                        .
                        .
                        .
File: @  Node: Basic,  Up: Top,  Previous: Top,  Next: Output

Fundamentals of @ for programs:
When you make a listing, you must initially specify the names of the
                        .
                        .
                        .
File: @ Node: Output, Previous: Basic, Up: Top, Next: Text

What @ Output for Programs Looks Like.
@ listings of programs can contain all these things:
   One or two title pages
   Page maps
   Subtitle table of contents
   Listed source
   Symbol table
   Cref table.

Exactly which are present in a given listing depends on the switches
used (*Note Switches: Switches.).

The TITLE PAGE is meant to identify the listing.  It's most prominent
                        .
                        .
                        .
The SYMBOL TABLE is a relatively dense alphabetical table of all
symbols and where (and how) they were defined.  Both single-file
symbol tables and "universal" (all files sorted together) symbol
tables can be requested.  If you make a cref, you might not want the
symbol table as well;  see the /$ switch.  In Lisp code, you may find
that a few long symbol or definition-type names cause the symbol table
to be printed using a very few wide columns.  The /nA switch specifies
that all names be truncated to only n characters, in printing the
symbol table.  This will cause more columns to be used and thus fewer
pages to be needed.

The CREF table is a list of all definitions and all references for all
symbols.  The symbols appear alphabetically, at most one on a line;
                        .
                        .
                        .
```

Fig. 9. Portion of Atsign's User Documentation: Symbol Table

deletion. The paragraph explaining the *symbol table* in the *output* section of Atsign's documentation is found and the programmer is able to add the last two sentences highlighted in Figure 9.

Since there are no further categories which were selected for this modification, the system can stop asking questions and wait for another modification to be made.

The *where* questions in this example are very helpful in finding the location of obsolete sections of user documentation. It is not as obvious why the system bothers to ask the *what* and *why* questions. These questions are useful, however, for many reasons.

For one, they summarize information about the development of a program. A record may be kept of the number of changes made which fall into each category. The tool would be able to tell how many changes performed a certain function or were made for a specific reason. This is particularly important in a large project where the supervisor must understand how numerous parts of a system are evolving and cannot study the effect of each modification.

Another way in which these questions may be helpful is in expanding the vocabulary of changes. An additional menu item, such as *other*, may be inserted in each question. When it is selected, the programmer would be asked to specify the new level, function, or reason.

The third reason for asking the *what* and *why* questions is to organize the thoughts of the programmer a little better so that he can write a more informative passage of documentation. Notice that in the scenario, the sentences added to the documentation (Figures 8 and 9) do mention the cleaning up of the symbol table's format and the extra functionality of the /nA switch. The *what* and *why* questions make the programmer think more about the change and how it affects the program. This may lead to more well-written documentation.

The maintenance of user documentation is one possible application of the taxonomy of types of changes. Other tools may be developed as a result of this or similar studies. System maintenance documentation gets obsolete just as quickly as user documentation. Perhaps a similar tool can be developed to keep it up-to-date. The categories of changes in the taxonomy are very general. There are many more specific characterizations lying below the surface which may be extracted by closer, in-depth studies on each level. This type of study may include an examination of the code as well as the modification history and user documentation.