Version Sensitive Editing: Change History as a Programming Tool

David L. Atkins

Bell Laboratories, Naperville, IL 60566, USA,
datkins@bell-labs.com
http://www.bell-labs.com/~datkins

Abstract. Software Version Control Systems (VCSs) are used to store the versions of program source code created throughout the software development cycle. The traditional purpose of such systems has been mostly administrative, providing the safe storage of source code and the ability to recreate earlier versions, as well as tracking the progress of new feature development and problem resolution. Software developers often regard the VCS as a necessary but unpleasant encumbrance to the software design and coding process. However, when the change history data gathered by the VCS is easily available to the programmer in the context of an editor, the programming process is enhanced. Faults introduced by earlier changes can be more rapidly located, dependencies on other changes can be avoided, and the version history provides valuable hindsight that can help to guide future development.

1 Introduction

Software development projects typically use a Version Control System (VCS) to store program source code and maintain the history of changes. Although the main purpose of the VCS is to provide administrative control over the code modifications and assure the reproducibility of past versions of the source code, the VCS is a rich repository of historical change information. Tools provided by the VCS may provide some access to this data, but usually the information is in the form of reports tracking the progress of feature development or problem resolution and does not show the details of the source code revisions. Recently more attention has been paid to visualizing software version history [Bal96] and interpreting source change patterns.[Bal97] These approaches mine the change history data for a great deal of information about the system as a whole and can be very useful as discovery and analysis tools. Using the revision history to perform a static postmortem of what has taken place during development can provide valuable insights into the software design and indicate areas for improvement.

We believe that an awareness of the software's origin at a detailed level can significantly improve the coding process. For example, knowing that the original code author regarded certain lines of change as a "quick and dirty fix" is an important factor in deciding how it should be changed. Or, knowing that some

B. Magnusson (Ed.): ECOOP 98, SCM-8, LNCS 1439, pp. 146–157, 1998.

lines of code have been untouched for several years while some other lines have changed in the past week certainly helps to locate a problem that has appeared in the last week. Being able to associate lines of code with their reasons for existence is almost like having the authors available to provide commentary on the software design.

The revision history is a valuable knowledge base and many of the questions about the code's vintage are most likely to occur as new changes are being made, i.e., while editing the program source. For example, to understand the past revisions to a file being edited with existing tools, the programmer first extracts two versions of the file from the VCS as plain text files, and then uses a tool such as Unix *diff* to compare the extracted files and reconstruct the change history. Such a procedure is inconvenient, requires the programmer to know which versions to extract, and only indirectly accesses information stored in the VCS. However, tight integration of the revision history and the editor could make the version data directly available at the right time in the right context, with an ease of access that encourages its use.

In the next section we discuss how a typical VCS stores version information and how that information relates to the coding activity. Section 3 describes the Version Editor, a tool that integrates editing with access to the VCS data. Section 4 illustrates the use of this tool in several scenarios. Finally, the last sections comment on the interface and implementation of the Version Editor and suggest areas for further investigation.

2 Version Control Data and the Coding Environment

Typical Version Control Systems require the programmer to retrieve (check out) a source file for editing. The programmer then modifies the file and returns (checks in) the revised file to the VCS, creating a delta to the file. The delta encapsulates the modification made to the file: the change itself (as the lines inserted and deleted), who made the change, and when. The VCS usually requires some sort of description of the change and may associate many deltas to one or more files with the same change description; tools are provided to extract versions of the source with a particular set of deltas applied to each file. The development process will often provide a mechanism for maintaining and updating the status of the modification as it moves through testing, final approval, and incorporation into a product. In this type of system, there is much information stored about each change.

At the file level, each line of a source file could be associated with the programmer who created the line, the date and time that the line was created, the description of the reason for the change, and the status of the change. Most of this information is buried deeply in the implementation of the VCS. For example, a VCS that uses the Source Code Control System (SCCS)[Roc75] or Revision Control System (RCS)[Tic85] for storage will have delta information about each file that could be used to determine who inserted and deleted each line, and when the delta was made. Figure 1 shows the three views of a file as it changes

over time and how the corresponding version history is stored by SCCS. In such a system, the deleted lines (i.e., those lines that no longer appear in the most current version of the file) are also stored. Although this information exists in the VCS, and is used to construct past versions of the program source, the information is not readily available to the programmer working on a current version of a file.

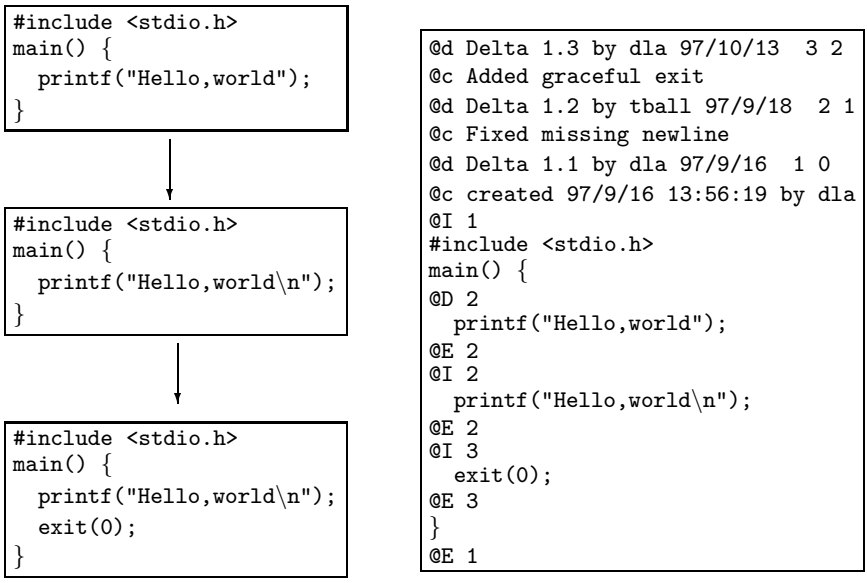

Fig. 1. Seqence of revisions and corresponding SCCS storage.

The typical coding process consists of iterations through a checkout-edit-checkin cycle, using any text editor to modify the source code retrieved from the VCS. The view of the code in the editor ignores change history information. That is, the working version of the source code contains no version information, it is simply text. In this environment, the programmer has no way of knowing if the lines of code being viewed were part of a bug fix or feature development, whether the lines were changed last week or two years ago, or in which versions of the product the code will appear.

However, knowing the origin and status of the source can be an essential factor in understanding the code and how it should be modified. Programmers often resort to comparing file versions with *diff* in order to understand what has changed recently and the development methodology may even require code inspections using listings that mark the code that has been changed. Many VCSs provide tools to annotate a version of a file with change data or perform a comparison between two versions. However, such tools are external to the actual editing activity and require the programmer to invoke the tool each time some

contextual version information is needed. The information must then be correlated manually to the lines of code being viewed in the text editor. Comparison tools also require that the programmer know which versions to compare, and that may not be readily apparent.

To be most useful to the programmer, the change history data must be easily accessible at the time it is most needed — while the programmer is looking at the code and deciding how to modify it. In addition to ease of access, the programmer should be able to view past versions of the code in the context of the current working version in order to fully comprehend the evolution of the program source.

An ideal software development environment would be able to provide the programmer with immediate answers to the following questions while the programmer is looking at existing lines of code and attempting to change them:

- What changes have I made so far?
- When was this line created?
- Why was this line created?
- Who created this line?
- What other lines were created for the same change?
- What lines were deleted by this change?
- What is the status of this change?
- What is the status of other lines surrounding my change?

Moreover, the answers should be available in the context in which they are asked. That is, answering the questions should not require a context shift to a separate tool from where the code is being edited. And, it should be possible to ask the questions easily with minimal training in the use of a new environment.

3 The Version Editor

The Version Editor (VE) is our implementation of such a version sensitive programming environment. VE is a replacement for a standard text editor but also has access to the version data stored by the VCS and can use that data to answer questions and control editing. This tool has been used over the last decade by many software developers in small to very large projects within Lucent Technologies. VE works with several SCCS-based VCSs used at Lucent, including SABLIME and the Extended Change Management System (ECMS)[Mid97] , as well as working with plain SCCS files. Since VE duplicates the interfaces of Vi and Emacs (the two most widely used screen editors in the Lucent development community), programmers are able to use the tool with little or no training. The rest of this section describes how VE integrates the VCS information into editing.

VE makes the change information about each line easily available. For a plain SCCS file, the delta information consists of the login of the user inserting the line, the date the line was inserted, the SCCS delta number, and the description of the delta if there is one. SABLIME and ECMS systems extend the version

information by using Modification Requests (MRs) to group together one or more deltas to one or more files with a brief description (abstract), a long description, and status (e.g., in-progress, submitted for testing, approved). For SABLIME and ECMS files, VE can display the MR associated with a line, the status of the MR, the date of the delta, who made the delta, and the abstract of the MR. VE displays this information about the current line in a status area of the display, either continuously, or on demand. Thus, for each line the programmer can instantly know the when, who, and why of the line's origin.

In its default configuration, VE provides the programmer with immediate visual feedback about the current delta. Lines that have been touched in the current editing session (since the file was checked out from the VCS) are highlighted (e.g., by bold face type or reverse video), allowing the user to always know what they have changed so far. This is particularly valuable when iterating over an edit-compile-test cycle since the programmer can see at a glance where the original version has been modified and the size of the change.

For example, Figure 2 shows how the example file looks in VE after the function main has been changed to return an integer value. VE highlights the second and fourth lines in bold since those are the lines touched by the change. The original second line was changed by adding "int" as a return type for the main function and the fourth line is a new line added to the original version. As the editing takes place, VE highlights the line to show visually that it has been changed in this editing cycle. Note also that the editor's cursor is sitting on line three and VE shows the version history for that line.

```
#include <stdio.h>
int main() {
  printf("Hello,world\n");
  return(0);
}

SID 1.2 by tball,97/9/18 [Fixed missing newline]
"main.c" [modified] line 3 of 5
```

Fig. 2. VE view with changed lines in bold

Not only does VE keep track of changed lines, but VE also remembers the original lines. Usually the programmer is most interested in the way the file looks now, with the revisions just made. However, in some cases, it helps to see the current changes in the context of the way the file looked before those changes. VE allows the editing view to be adjusted so that deleted lines can be seen along with the newly inserted lines, where the deleted lines are distinguished by display attributes, e.g., underlining. In such a view, VE is careful to prohibit the deleted lines from being edited since they no longer exist in the checked out version of

the file, and are only shown by VE for informational purposes. However, deleted lines may be copied, which is especially handy when portions of "old" code need to be resurrected.

The ability to display deleted lines is not restricted to the current delta, but can be controlled by a variety of version attributes stored by the VCS. In the extreme, all deleted lines could be made visible in VE, permitting a view of the complete evolution of the file.

VE also has a command to "un-delete" a line that has been deleted in the current session. This allows fine control over the editing activity and preserves the importance of the current delta highlighting. Note that un-deleting is different from the standard undo feature of many editors since any deletion of a line in the current session may be undone regardless of any editing activity that has taken place since the deletion.

The version attributes for each line are used for several purposes in VE. As noted above, lines deleted by the current delta can be made visible. In general, the visibility and visual presentation (bold, underline, color, etc.) of a line is determined by its version attributes. Although the default view corresponds to the checked out version of the file with the current edits highlighted, the programmer can adjust the visibility and display criteria to produce a variety of views. For example, highlighting could be used to attract attention to all lines inserted after a particular date, or all lines whose change status is not yet approved by testers, or all lines touched by a particular set of changes. The view can be adjusted while the programmer is editing in VE to dynamically alter the viewing context as questions arise. The same criteria can be used to view multiple files, or each file may have its own interesting view, and the view can be toggled easily to allow comparison of versions.

In addition to affecting the display, version attributes can be used to perform searches in VE. For example, when the cursor is sitting on a line, a simple editor command can advance the cursor to the next line with the same version attributes, i.e., the next line inserted by the same delta. In general, version attributes can be used to perform searches much as an ordinary editor allows pattern searches. The programmer could search to the next line inserted after a certain date, by a specific programmer, associated with a particular MR, or any Boolean combination of these and other attributes. Thus the file may be navigated on the basis of its change history.

4 Example Coding Scenarios

In this section we present two simple examples of using the Version Editor to improve the code editing process. These examples were extrapolated from real coding experiences using VE.

In the first scenario, assume that we have a large program that performs many file manipulations. The program has been working well, but has recently developed a problem with too many files being left open. To track down this problem, all sections of the code that open files must be checked to make sure

that the open files are eventually closed. Various code browsing tools can be used to locate the relevant areas of the program source, but careful code examination is required to identify the problem. Figure 3 shows a normal editing view of one of many instances in the program where files are opened and closed. At first glance, the code looks to be correct, with the directory opened and then closed.

```
String FindSource(String base, String dir) {
   DIR * dirp = opendir(dir);
   for (int i = 0; i < NS; ++i) {      // Loop over suffix list
      String tmp = base + suffix[i];   // Target name to find
      for (dirent *de = readdir(dirp); de != NULL; de = readdir(dirp))
         if (tmp == de->d_name) {     // We found it, stop looking
            return tmp;
         }
      rewinddir(dirp);
   }
   closedir(dirp);
   return "";      // No match was found
}

"findsource.c", line 13 of 19
```

Fig. 3. Plain vi view of possible problem code

However, knowing that the problem has appeared recently is a valuable clue. By using VE to highlight the recent changes as in Figure 4, the problem area can be identified much more quickly. In this view, the lines deleted in the last month are made visible and are underlined, and the lines added in the last month display in boldface. When we examine the function that is opening a directory and traversing it in this view, we immediately see that a recent change causes the function to return from a loop, where it had previously broken out of the loop with control flow progressing to the normal function return at the end. Since the open directory is only closed outside of the loop, the change from "break" to "return" means that the directory is never closed.

It is certainly possible that by studying the code carefully, we would eventually notice this control flow problem. However, given that we may have dozens of such code fragments to examine, finding the problem would take much longer. VE's highlighting shows us instantly that there has been a recent change to this code, so it warrants closer scrutiny. The in-context display of the actual change facilitates an understanding of the effect of the change relative to the problem we are investigating, permitting the error to be quickly recognized and fixed.

```
String FindSource(String base, String dir) {
  DIR * dirp = opendir(dir);
  String result;    // The filename, if found
  for (int i = 0; i < NS; ++i) {     // Loop over suffix list
    String tmp = base + suffix[i];   // Target name to find
    for (dirent *de = readdir(dirp); de != NULL; de = readdir(dirp))
      if (tmp == de->d_name) {    // We found it, stop looking
        result = tmp;
        break;
        return tmp;
      }
    rewinddir(dirp);
  }
  closedir(dirp);
  return result; // Return the found name (may be null)
  return "";    // No match was found
}

Deleted by MR 595 by vz,97/11/15,approved [Stop source search at 1st match]
MR 467 by dla,97/9/21,integrated [Find source using list of suffixes]
"findsource.c", line 15 of 23
```

Fig. 4. VE view with recent deletions underlined, additions bold

Our second example scenario is taken from a real development environment where many developers make changes in parallel to the code, and the order of inclusion of the changes into product versions does not necessarily match the order in which the changes were made.

In this example, we need to make a change to a function that is called when a file is closed in a version editor. As originally designed, the function to be changed just releases the resources used for the file buffer in the editor: the version table object and the array of Line objects. Testing and investigation have shown that only the storage for the pointers to the Line objects is being released, not the storage for the Line objects themselves. Our task is to modify the function to properly release the Line objects. A natural implementation would be to loop over the Line objects, releasing each one in turn.

When we use VE to edit the function, we see the view presented in Figure 5 and find exactly the loop that we need. However, in this view, other developer's changes which are not yet approved for the official version of the product are highlighted in italics. This highlighting shows us that the loop over the Line objects is code that is still under development — it is part of a change that has not yet been submitted for inclusion in the final product.

Knowing that the code we see highlighted is not yet approved is very important information in our decision about whether to use the loop. If we add code within the loop, our new change will be dependent on the unapproved (and not even submitted) change. That is, it will be necessary to include both changes in order for our change to compile and work properly. Since VE shows us the

```
extern Version *vTable;  // The version table
extern Line * lines[];       // Array of pointers to the line objects

void CloseTheFile()
{
  // Calculate lines inserted, deleted
  int inserted = 0, deleted = 0;
  for (int i = 0; i ¡ nlines; ++i) {
    if (lines[i]->InsertingDelta() == vTable->CurrentDelta())
      ++inserted;
    if (lines[i]->DeletingDelta() == vTable->CurrentDelta())
      ++deleted;
  }
  cerr << inserted << " lines inserted, " << deleted << " deleted";

  // Release resources for the file
  delete vTable;
  delete [] lines;
}

MR 526 by tball,97/11/11,assigned [Report size of change at file close]
"close.c"  line 10 of 26
```

Fig. 5. VE view with unapproved changes highlighted in italics

author of the other change, we can contact the author to find out if his change will be submitted soon. If not, we must be careful to keep our implementation from becoming dependent on his change by coding our own loop to traverse the array of lines. In any case, the knowledge about the status of other code is important in our coding decisions. Having this information available as we are editing permits us to adjust our coding appropriately, and avoids unpleasant surprises about dependencies that we would otherwise encounter after the completion of coding.

5 Notes About the Interface and Implementation

One of the most important principles guiding the design of the Version Editor was ease of use. In order for the tool to be accepted, developers must be able to use it in existing environments with no re-training. So, rather than design a new editing interface, VE very faithfully emulates existing editing interfaces. This allows new users to immediately begin to edit with VE (possibly without even realizing it is not their normal editor). Some of the extra functionality of VE can be made available for "free". E.g., the highlighting of the current changes just happens without requiring any user action. Another example is the standard Vi command (Control-G) to display the current line number. In VE, the same command not only displays the line number, but also displays the line's change history: originator, date, and MR abstract. Other commands for version specific functionality are modeled after standard editor commands. For example, version

search semantics are like standard text search semantics, allowing forward and backward searches and remembering the object of the last search.

The Version Editor can even be used with ordinary text files so that users do not have to shift tool sets. Of course, for an ordinary file there is no version history, but editing can still benefit from the highlighting of the current changes.

The Version Editor accesses the underlying SCCS files of the VCS directly so that it has all available data about the change history of the file. Additional change information maintained by the VCS is also correlated to the delta information stored by SCCS. Each line in a file has an associated set of version attributes. These attributes consist of the delta that inserted the line and zero or more deltas that deleted the line. Each delta in turn has a set of attributes: when it was made, by whom, a possible delta comment, and a corresponding MR (for SABLIME and ECMS VCSs). The MR then has its own set of attributes: the status of the MR, when the status last changed, and the abstract of the MR. Taken together, each line of the file has a rich set of version attributes.

The "version" of a line is its delta set. This notion of version enables VE to compare lines for their version so that all lines changed at the same time may be grouped together and meaningful views of the file can be created. However, since all attributes directly or indirectly associated with a line may be used to control visibility, artificial views of the file (e.g., showing both the old and new versions of a line) may be created.

VE re-implements the Vi and Emacs text editors. Although this may at first seem unnecessary, the version information is needed at such a low level in editing that performance might not be acceptable if VE were implemented as a layer on top of a plain text editor. In particular, every keystroke would have to be intercepted as well as every display update by the plain editor. Since VE was initially designed, more powerful programmable editors such as GNU Emacs[Gli97] are now available, and it is possible that such an editor could be extended with knowledge of the VCS to provide version sensitive editing like VE.

6 Related Work

The term version is often used to describe two different aspects of variations of a source file. It may refer to flavors of the file extracted for different purposes, e.g., different code constructs needed for distinct operating system platforms. The term also may refer to the sequence of changes over time as the source code is developed. The editor P-EDIT[Kru84] addresses the problem of dealing with multiple code variants. This tool hides the variant selection constructs and allows the user to view and edit different versions within a single editing session. The Version Editor provides similar capabilities by automating the creation of the variant control constructs of ECMS that are embedded in the source and these capabilities have been described elsewhere[Pal89] . However, P-EDIT does not address the problem of dealing with changes over time and interfacing to the VCS. Some editors (e.g., GNU Emacs[Cam96]) provide an interface to the VCS.

However, this interface is a convenience limited to executing VCS commands with the editor's buffer and does not integrate the change history into the editor. Microsoft Word[Cat97] implements some degree of revision control, but this is restricted to the file edited and does not incorporate any information from an external VCS. The editor EH[Fra87] provides sophisticated management of the version tree but does not deal with accessing the change history of a file stored under a conventional VCS.

7 Summary

Version Control Systems can be a rich source of information about the history of change that has led to the current state of a program's source. Knowledge of this change history can improve the understanding of the code and aid in the design of enhancements and problem solutions. Programmers often ignore the version data because it requires too much effort or it is too distracting to unearth the history during the coding process. By integrating the version data into the code editing environment, the information can be made available when it is likely to be of the most use.

The Version Editor is a tool that puts the change history into the editor where it can be instantly accessed and used to control editing and convey version information in the editor's display. The easy availability and disposition of the change history benefits the coding process. Problem areas in code may be identified more readily, unwanted dependencies may be avoided, and the version dimension can increase comprehension of code.

The Version Editor was originally designed to work with existing SCCS based VCSs whose purpose is the administration and safe storage of program source. The VCS only stores changes at the line level using a file comparison algorithm, but the usefulness of the version information during editing suggests that it would be more desirable for a VCS to capture more of the editing intent. That is, a VCS that stored a finer unit of change that corresponds to the actual editing could provide even more version context to enhance the coding process.

8 Acknowledgments

The Version Editor traces its roots to early work by Coplien on an experimental Delta Editor[Cop87] . The initial implementation with SCCS files was done by Maria Thompson and Pat Baldwin. Anil Pal collaborated on much of the design of the Version Editor and Joe Steffen provided the implementation of the Emacs interface. Special thanks are due to Tom Ball and Audris Mockus for all of their suggestions about this paper.

References

Bal96. Thomas Ball and Stephen G. Eick, Software Visualization in the Large. *IEEE Computer*, 29(4):33–43, April 1996.

Bal97. T. Ball, J.-M. Kim, A. Porter, and H. Siy, If Your Version Control System Could Talk... , *ICSE '97 Workshop on Modeling and Empirical Studies of Software Engineering*, May 1997.

Cam96. Debra Cameron, Bill Rosenblatt, and Eric Raymond, *Learning GNU Emacs*, O'Reilly & Associates, 1996.

Cat97. Catapult, Inc., *Microsoft Word 97 Step by Step, Advanced Topics*, Microsoft Press, 1997.

Cop87. J. O. Coplien, D. L. DeBruler, and M. B. Thompson, The Delta System: A Nontraditional Approach to Software Version Management, *AT&T Technical Papers*, International Switching Symposium, March 1987.

Fra87. Christopher W. Fraser and Eugene W. Myer, An Editor for Revision Control, *ACM Transactions on Programming Languages and Systems*, 9(2), April 1987.

Gli97. Bob Glickstein, *Writing GNU Emacs Extensions*, O'Reilly & Associates, 1997.

Kru84. Vincent Kruskal, Managing multi-version programs with an editor, *IBM Journal of Research and Development*, 28(1), January 1984.

Mid97. Anil K. Midha, Software Configuration Management for the 21st Century, *Bell Labs Technical Journal*, 2(1):154–165, Winter 1997.

Pal89. Anil Pal and Maria Thompson, An Advanced Interface to a Switching Software Version Management System, *Seventh International Conference on Software Engineering for Telecommunications Switching Systems*, Bournemouth, UK, July 1989.

Roc75. Marc J. Rochkind, The Source Code Control System, *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

Tic85. Walter F. Tichy, RCS — A System for Version Control, *Software — Practice and Experience*, 15(7):637–654, July 1985.