TEXT DEVELOPMENT AND MANAGEMENT IN UNIX-BASED PROJECTS

Robert J. Glushko

Bell Laboratories Whippany, NJ 07981

ABSTRACT

Mismatches in technology, media, and organizations in the development of software and software documentation often leave the documentation late, incomplete, and poorly coordinated with the software. One solution to this problem is to treat software documentation like software. SOLID, the System for On-Line Information Development, builds on the popular UNIX* operating system and exploits the idea that all information that can be stored on a computer shares a common life cycle and can be created, managed, and delivered with the same methods and commands. SOLID, together with the UNIX system, solves ten key problems of text development and management that can be faced by any project.

INTRODUCTION

The Problem: Mismatch Between Software and Documentation

As computer systems become increasingly complex, more comprehensive and timely documentation is needed to support them. Unfortunately, the methods for developing and delivering software documents have not kept pace with the rapid advances in software technology that have made the systems possible. New operating systems, programming languages, and software development tools have made programmers more productive, but few comparable breakthroughs have helped document writers. Much of the documentation for the latest computer systems is produced and delivered on paper by methods that have changed little in decades. This mismatch between the technology and media of software and documentation often leaves documentation late, incomplete, and poorly coordinated with the software that it is supposed to serve.

The mismatch between documentation and software often includes a mismatch in the organizations that produce them. Instead of working closely with software developers from the start of a project, document developers may have little contact with the system or the developers until the software is nearly complete. Document developers and programmers usually work in separate, non-parallel organizations, impeding the communication and interchange that hastens and clarifies documents. When software changes, the corresponding changes to the documentation may not be made for months, if they are made at all.

Word Processing Is Not the Solution

Many software projects have tried to solve the problems of late and incomplete documentation by speeding up document development using word processing. But it solves only part of the problem. Word processing's limitations can best be seen by contrasting typical word processing work with software documents.

Most word processing systems support the entry, editing, formatting, and filing of documents. Such documents typically have single authors, undergo little revision, and need no coordination with other documents during their development or delivery.

Contrast these documents with the documentation for computer systems, including user and operator manuals, training materials, etc. Such documents are often lengthy, have many authors and versions, and require complex formatting. In addition, they need serious source and configuration controls to prevent simultaneous editing, to allow for backups, and to synchronize named releases in coordination with software.

Word processing may be useful as a tool for an individual writer when its metaphor of the video screen as an electronic piece of paper--usually described as "what you see is what you get"--is appropriate. But in document development for complex computer systems, what we see isn't all we want. What we

---

* UNIX is a trademark of Bell Laboratories. My comments refer to the version known as System V distributed by Western Electric.

CH2017-2/84/0000/0473$01.00 © 1984 IEEE

want is coordination between software and its associated documentation throughout the complete life cycle of the system.

The Solution: Treat Documentation Like Software

With others at Bell Laboratories,** I have attempted to exploit a different metaphor for the development of software documents--software documents are like software. The text of programs is read by computers, and the text of documents is read by people, but both can be created, edited, managed, and distributed in the same time frame, in the same medium, and using the same tools.

Our goal was more than just copying a "programmer's environment" to create a "documenter's environment." We set out to borrow and generalize the tools and methods of software development to create a common information repository for all the information needed by everyone involved in a project. System designers, human factors engineers, technical writers, programmers, and project managers would all work in a common information database that contained the software, documentation, and other text associated with the system throughout its entire life cycle.

BUILDING ON UNIX

We started with the UNIX operating system. The UNIX programming environment is widely used, especially because of its emphasis on software tools, its programmable command interpreter, and its hierarchical file system.[1] An important collection of UNIX utilities is known as the Programmer's Workbench, upon which developers build their own customized development environments.[2] In addition, the UNIX system has significant text processing utilities, including commands for pattern searching in files, sorting, counting, and editors and formatters.[3] Our goal was to build a portable framework for information development that integrated all the text development and management activities, while overtly supporting the customization needed to optimize it for a specific project.

SOLID--System for On-Line Information Development.

We built several system development environments and studied those built by others. We analyzed the goals each had, the concepts they embodied, and the implementation details. Finally, we put together SOLID, the System for On-Line Information Development.[4] The name contains neither "Software" nor "Documentation" because the core principle is that both are information in text form that have a common life cycle.

SOLID provides commands for outlining a new source module, editing it, making it "official," extracting a copy to update, reinstalling the revisions, and then generating and storing the final product from the source. SOLID contains a simple on-line documentation system from which formatted documents can be retrieved by name, the date they were last changed, or by keywords contained in titles or section headings.

SOLID can be thought of as consisting of two domains. Source files (editable documents or program text) constitute one domain, and product files (formatted documents, executable programs) make up the other. The source domain is subdivided into types. Types are groups of source files that share a common outline or skeleton, a programming or formatting language, or a membership in a sub-system. Finally, the source domain is mapped to the product domain by generation procedures for each type.

For example, the source type cmain contains C language program source files. The generation procedure finds the source file in the appropriate directory in the source domain for that type, runs the C compiler with all the necessary options, and installs the executable binary file in the product directory that contains executable commands. The source type doc/cmd contains editable versions of UNIX-style reference documents for user commands from which the nroff text formatter generates on-line documents ("manual pages").

Organizing information into types provides the flexibility to create, maintain, and deliver all text information in the project without influencing design decisions. Type definition is completely under administrator control and specified in tables for visibility and extensibility. In general, all SOLID's software and documentation is under SOLID control, so the SOLID administrator can easily change or extend any of the system's capabilities.

We started sharing SOLID informally within Bell Labs in late 1981. Our user community now contains dozens of sites throughout Bell Labs and Western Electric where it is used to develop software documentation, programs, computer-based training, and combinations of all three. Each site supports as many as thirty simultaneous users.

PROBLEMS OF TEXT DEVELOPMENT AND MANAGEMENT

In the sections that follow, ten problems of text development and management that are solved by SOLID and the UNIX system are described. Each problem is characterized by a few colloquial symptoms that show that the problems are general. Where appropriate, the UNIX

** Especially Mike Bianchi, Tom Foregger, Ken Hicks, John Mashey, and Pat Parseghian.

tools that contribute to the solution are discussed.

Shared Development.

- You edited that, too! Now I have to edit YOUR copy to add the changes I just made to MY copy.
- I know someone is editing that now, but I don't know who it is.

Any project with significant text to develop must keep a master copy of the information and prevent simultaneous editing. Most UNIX-based projects use the Source Code Control System[4] to keep track of multiple versions and to control when and by whom changes can be made. SOLID's interface to the UNIX SCCS utilities like **admin**, **delta**, and **get** uses the "source type" abstraction to hide the details of where files are stored. SOLID also makes it easy to incorporate local procedures into the source control scheme; for example, to inform software developers automatically, via electronic mail, whenever a requirement document changes requires little more than an entry in a table.

Version Control.

- Is this the latest version?
- When did it start doing THAT?
- Who made it do THAT, and why?
- I know it worked last March.
- What "Verbose" option? There's nothing about it in the manual.

SCCS efficiently stores the complete history of a source file by recording only the changes from one version to the next. Any version of a source file can be reconstructed by starting with the first version and executing as many intermediate changes as necessary. The date and version of any piece of product that was generated from SCCS-managed source can be determined using the UNIX **what** command.

The last symptom of this problem is not so easily cured, but "synchronizing" software and documentation is made possible by a uniform view of both kinds of source. Documentation becomes more important and visible when it is treated just like software and developed at the same time in the same environment.

Redundant Text.

- The overview section should be the same in the user, administrator, and management manual.
- Those two paragraphs were copied all through the user manual. It will take weeks to find and change them all.

SOLID's "collection" mechanism includes the same document in different logical units, such as guides for different users of the system, without making additional copies of the file. Collections appear to users as custom tables of contents that list only those on-line documents relevant to their jobs. If the document modules can be used repeatedly, but are too small to make sense by themselves, SOLID manages them as "document fragments." Fragments are inserted into documents using the **nroff** text formatter's "include" request.

Information Retrieval.

- I know it is in the manual somewhere.
- Is that document available yet? The table of contents hasn't been updated in weeks.
- Is this copy the latest version?

The ancestor of SOLID's on-line document retrieval system is the standard UNIX command **man**, which retrieves reference documents from the on-line user manual.[4] Searching for documents whose names, titles, or section headings contain keywords is a straightforward use of the UNIX pattern search command **grep**.

SOLID automates the tedious tasks of inventorying and indexing the on-line documents. Every night, when the load on the computer is lightest, programs scheduled by the UNIX **cron** program determine if any documents have been added or changed and updates the tables of contents and indexes if necessary.

Standards

- Why don't people follow standards around here?
- Have you seen the standards manual? It is 100 pages long!

Often documents have a well-defined structure with a standard format. For example, documents in the UNIX user's manual have sections for NAME, SYNOPSIS, DESCRIPTION, EXAMPLES, etc. An easy way to create a new document of this type is to start with an outline or "skeleton" document and then "fill in the blanks." SOLID generalizes this basic idea so that any source type has a skeleton that guides the developers to what information is needed while painlessly enforcing standards. Skeletons help users as well, because consistent style assures them that related documents and programs contain similar content in standard form.

SOLID users can invoke the commands of the UNIX Writer's Workbench[7] to find spelling mistakes, punctuation errors, split infinitives, and awkward wording or sentence structure.

Usability

- How do I get started?
- What can I do next?
- What should I do next?

This problem was challenging to solve. The features that make the UNIX system "programmer-friendly," the terseness and modularity of its commands, make it difficult for non-programmers to use. SOLID has both a UNIX-style command interface and a menu interface that supports task-oriented menus, extensive prompts and feedback, and context-sensitive help.

Regeneration of product

- Our guru isn't here, and now nothing compiles (formats).
- No one knows how all the pieces fit together.

The generation procedure, the instructions for translating a source file into its product form, is a necessary part of the definition of a new source type in SOLID. Generation procedures in SOLID are under source control, just like every other type of text, so it is impossible to "forget" how the pieces fit together. The UNIX **make** command is one kind of generation procedure.

Updates

- This page intentionally left blank.
- Pages 4A, 4B, 4C?
- But we haven't merged the last 3 updates yet!

A popular rule in software engineering is that "one module should hide one secret." By treating software documents just like software, SOLID encourages modular documentation, in which "one document should tell one secret." Modularity keeps documents small, makes them easier to write and manage, and limits the scope of changes. There is no need to accumulate many changes to justify reissuing a bulky manual.

Multiple development sites

- You don't have that? We fixed that 3 months ago?
- New York and LA have the same versions. Don't they?
- The tape (document) is in the mail.

SOLID commands for delivering and installing source files can support networks of systems for multiple-machine projects. The UNIX uucp ("unix-to-unix copy") command manages the file transfer. A computer running SOLID might serve as a single target for information delivery from many suppliers. It could then serve as the development environment in which local information can be integrated with the supplied information. Finally, it might function as an "electronic library" to distribute documents to end users.

Portability

- File names are buried in every program. We'll never get the system to run on the new computer.
- But it only runs on XYZ terminals with options A, C, and F.

SOLID runs on "standard" UNIX and is written using the C programming language and the UNIX shell command interpreter, so as UNIX becomes available on new or different processors, SOLID does too. SOLID commands rely on generic place names by using shell environmental variables. The menu interface is built using **curses** and **terminfo**, a UNIX virtual terminal toolkit that makes SOLID's code work without change on any terminal with cursor control functions.

SUMMARY

It is not surprising that treating documentation like software produces better documentation. Projects that use SOLID report that documents are more timely, more complete, and better coordinated with software. What is somewhat surprising is that treating documentation like software also produces better software. Programmers have earlier and more reliable requirements, and the closer involvement of non-programmers throughout the software life cycle leads to better design and more rigorous testing.

In building a text development and management system using the UNIX programming environment, we took advantage of existing UNIX tools. We preserved the "toolkit" philosophy so that SOLID can make use of new UNIX utilities as they become available, especially since experience has often shown that programs not intended for document preparation can make it easier. Our original charter had been to support document delivery in telephone company operations by "piggy-backing" onto a time-shared UNIX management information system.[8] Thus, SOLID was not designed to be a full-fledged publication system, and does not take advantage of bit-mapped display terminals. At the same time, however, these restrictions to a "computer-center" architecture and ASCII terminals have made the system less complex and more portable. Since we designed SOLID by studying how projects customized the UNIX programming environment, it is not surprising that SOLID fits a wide range of development needs.

REFERENCES

[1] B. W. Kernighan and J. R. Mashey, "The UNIX Programming Environment," Computer, vol. 14, no. 4, pp. 12-24, April 1981.

[2] T. A. Dolotta, R. C. Haight, and J. R. Mashey, "UNIX Time-Sharing System: The Programmer's Workbench," Bell System Technical Journal, vol. 57, no. 6, part 2, pp. 2177-2200, July-August 1978.

[3] R. Furata, J. Scofield, and A. Shaw, "Document Formatting Systems: Survey, Concepts, and Issues," ACM Computing Surveys, vol. 14 no. 3, pp. 417-472, September 1982.

[4] M. H. Bianchi, R. J. Glushko, and J. R. Mashey, "A Software/Documentation Development Environment Built From The UNIX Toolkit," In H.-J. Schneider and A.I. Wasserman (Eds), Automated Tools For Information Systems Design. Amsterdam: North Holland, 1982, pp. 107-108.

[5] M. J. Rochkind, "The Source Code Control System," IEEE Transactions of Software Engineering, SE-1, pp. 364-3/0, December 1975.

[6] R. J. Glushko, "Lessons In The Evolution Of A Document Retrieval System," Proceedings of the 44th Annual Meeting of the American Society for Information Science, pp. 237-239, October 1981.

[7] N. H. MacDonald, "The UNIX Writer's Workbench Software: Rationale and Design," Bell System Technical Journal, vol. 62, no. 6, part 3, pp. 1891-1908, July-August 1983.

[8] R. J. Glushko and M. H. Bianchi, "On-Line Documentation: Mechanizing Development, Delivery, And Use," Bell System Technical Journal, vol. 61, no. 6, part 2, pp. 1313-1323, July-August 1982.