Cleaning Up UNIX® Source
- or -
Bringing Discipline to Anarchy

David Tilbrook
Zalman Stern

Information Technology Center
Carnegie Mellon University
dt@andrew.cmu.edu
zs01@andrew.cmu.edu

ABSTRACT

Many people are coming to the realization that the distribution of UNIX software is not just a matter of creating a tar tape. Approaches to software and distribution management vary from simple disciplines, conventions and procedures to full scale Integrated Programming Support Environments (IPSEs). This paper discusses some of the problems and issues involved in the management of software, concentrating primarily on the exporting of source to remote locations.

1. Introduction

This paper is a discussion of techniques for managing medium size software distributions (e.g. 4.3bsd[1]) and for the installation of that software at remote locations. For the most part, the paper is based on the strategy originally formulated by David Tilbrook to manage and distribute the D-Tree software [Tilbrook 86]. In this paper the authors attempt to extract the relevant points that should be applicable to most UNIX based software projects. The Information Technology Center plans to use this strategy to manage the Andrew software produced at Carnegie Mellon University. If this endeavor is successful, these techniques will be applied to other software developed at CMU as part of the effort to produce a CMU distribution tape.

The paper is divided into eight sections. The next two sections introduce the goals and objectives of Software Mangement and some of the guidelines to be followed in implementing a Software Management system. These sections are followed by a discussion of some of the more important aspects of such an implementation, namely the key personnel and their responsibilities, the contents and structure of the supporting database, and some of the more important tools. The final section presents some prognostications as to the effectiveness of the strategy and its future.

2. What is Software Management

''Software Management'' is the discipline of managing, maintaining, and distributing a body of software in source form. The ultimate objective is unremarkable and painless installation of software and its subsequent upgrades at a remote site. The notion behind this is that software should be judged on its inherent worth, not the amount of hassle involved in getting it to run.

Software Management is a different discipline from Software Engineering. While the primary goal of both Software Management and Software Engineering is increased reliability of the subject software, the disciplines have very different approaches to achieving this goal. The software engineer is primarily concerned with creating, transforming and validating descriptions of a system. The software manager, in contrast, is primarily concerned with the management of those descriptions so as to make them available to other

[1] Approximately 4300 files and 850,000 lines of code.

parties. Furthermore, the software engineer should be concerned with the elimination of semantic errors (e.g., bugs), whereas the software manager tries to ensure that the system, when installed remotely, is semantically equivalent to the system as delivered by the supplier.

Achieving software reliability through Software Engineering is largely a matter of the application of techniques to facilitate the creation, modification, transformation and validation of system descriptions. For example, the use of programming conventions, high level languages, and formal design systems all enhance the clarity of the descriptions and might facilitate the use of formal validation techniques.

Achieving software reliability through Software Management is largely a matter of providing consistent representation and control of the software source. The provision of such a representation and the controlling mechanisms should facilitate the rectification of problems and the distribution of the solutions in a timely and well-controlled manner.

The rest of this section outlines some considerations when designing a source representation.

- The representation must permit the easy extraction of a subset of the software for transmission to another location. The extraction mechanism must support the specification of the files to be extracted in a variety of ways, such as version numbers, subset specifications, or a list of changed files since the last appropriate extraction.
- The organization [sic][2] of the source must be done in such a manner as to allow personnel to manage, install and test the source without an extensive understanding of the software's purpose or function.
- The addition and/or removal of source files must be simple, not requiring extensive modification of any secondary files (e.g., system construction files). Ideally, the mere act of adding or deleting a file will cause the software construction tools to do the right thing.
- The procedures to create and install a distribution must be usable by a large number of people. If a software distribution is to succeed it cannot require or demand extensive understanding by the installer. To do so would impose limitations on the number of installations that could be performed by demanding more resources than are either merited or available.
- Solutions to distribution or installation problems must be applied across the entire source database in a consistent and easily-stated manner. For example, the mechanisms used to specify installation controls (e.g., installed file attribute specification), environmental variations (e.g., file naming mechanisms), search paths, versions of compilers to be used, include paths, and library paths must be expressed in a way that they can be easily modified by the user. Any significant change in the values of those controls has to force the re-creation and installation of any affected files. Achieving this objective is important since minimizing the mechanisms used to specify such controls will increase the flexibility of the installation and facilitate the adaptation of the distribution to different requirements (e.g., different compilers, operating systems, or machines).

3. General Principles

The principles described in this paper are guided by two considerations. On the one hand, the installation process should proceed flawlessly. On the other hand, there is Murphy's law. Despite all efforts to create a fail-safe distribution, installation will take place in an alien environment, which makes it likely to fail in new and different ways.

"Keep It Simple, Stupid" (a.k.a. the KISS principle). Another way of stating this is "Don't do what you don't understand." The area of Software Management is new and full of hard problems. The goal is to increase the reliability of software, especially at a remote site, not to solve all the problems in the field. Clients will not appreciate research being done on their systems. Therefore, the distribution system should be based on proven, well understood tools. Furthermore, if the software management system is too complex to understand, it will not be used successfully. On the one hand, to try to keep things simple, specialized tools should be avoided. Including such tools in the distribution can be very costly. On the other hand, a specialized tool will often greatly simplify a task by centralizing all of that task's complexity in one place. A good tool must fit well into the distribution as a whole and place minimal demands on the software manager. Attempts to solve the software distribution by writing more complex versions of *make*(1) are

[2] Since both authors are now being paid in American currency, some concessions have been made.

distressing in that they require very complex human generated Makefiles. In effect, they address the wrong problem, thereby adding many complex Makefiles throughout the distribution.

"If it won't take off in Poughkeepsie, don't expect it to land in East Podunk!" The first clue that a mechanism is too complex is that it does not work reliably in the development environment. If it is the least bit difficult for the software management staff to use a tool, it will be a failure in the installation environment. Makefiles which seem to employ magic are an excellent example of this phenomena. Many people struggle to make a Makefile work without realizing how fragile it is. Such "solutions" will only cause problems when they leave the development organization.

"Too many cooks spoil the broth." If a Software Management system requires more than one "cook", it is unlikely to achieve some of the goals discussed earlier. It must be designed and implemented to be completely manageable by a single person. This is a key point that will be discussed at length in the next section.

4. Roles and Responsibilities

Any Software Management system must accommodate three distinct responsibility classifications: the software gatekeeper, the software supplier, and the client site installer (i.e., the client). There are other types of personnel associated with the distribution (e.g., the end-user is conspicuously absent from the above list). However, at this time, we are concerned with the above three classes only.

4.1. The Gatekeeper

One of the guidelines established in the previous section was that a single person be able to manage all of the software. To ensure consistency, to facilitate communication and to centralize efforts, there must be one well identified individual who is assigned the authority and responsibility to manage and protect the software database. This person is called the **gatekeeper**.

The gatekeeper stands between the supplier and client, protecting both parties' interests. When examined with respect to the goal of increased software reliability, the gatekeeper's primary responsibility is to be the best source of the best information about the distribution's status, availability and compatibility. Hence her primary responsibility is to maintain and protect the integrity of the software database, and to ensure that the information required to install, use and maintain the subject software is available, accurate and (we hope) useful.

The gatekeeper need not be a programmer. Her responsibility is the management of source, not the creation of it. Understanding its functionality might be useful, but is not necessary. She should have a technical and administrative support group to aid in the preparation and validation of the database and to process the details of exporting and monitoring distributions to remote clients. She will also require the support of an "editorial board" which will act as the "arbitrators of taste." This board is a group of senior technical advisors and management who can advise the gatekeeper on the suitability or necessity for inclusion of a software package in the distribution.

The gatekeeper's responsibilities do not include component testing. She should expect and demand that software submitted for publication has been tested to an acceptable standard by the software supplier or some other party on their behalf. The gatekeeper will test the system, but only with respect to its construction and installation within the context of the entire distribution. Ideally, the gatekeeper should have access to a secondary group to do quality assurance testing, but this is not essential.

In order to fulfill her responsibility, the gatekeeper will have to elicit the cooperation of both the supplier and, eventually, the client, as covered in the following sections.

4.2. The Software Supplier

For the most part, it is assumed that the supplier is accessible to the gatekeeper, in that the latter may call upon the supplier to provide extra information and help where the delivered source is insufficient. In such cases it is also assumed that the software supplier will have sufficient interest in the proper distribution of their software to incorporate the changes required to make it conform to the structure and style of the database. If this is not the case, the gatekeeper may choose to reject software or to acquire alternative assistance to reshape the software as she requires. The point is that the supplier must accept the following realities:

- Their role is to deliver software in its complete source form to the gatekeeper along with the additional construction information in an acceptable or agreed upon manner (“read the Makefile” is not acceptable).
- Once it is delivered to the gatekeeper, the gatekeeper is in control of that version of the software.
- The gatekeeper can expect (and sometimes demand) the supplier to be prepared to rectify problems in previously released versions (up to an acceptable level) so as to be able to provide minor fixes to the client where a full upgrade is either impossible or unacceptable to the client.
- The supplier should attempt to inform the gatekeeper of any substantial activity that might affect the delivery of future releases of the software.

It is difficult to imagine a programmer who would reject such conditions, but it is rumoured [sic][3] that some of the breed have been known to be uncooperative on rare occasions.

The gatekeeper, in turn, will ensure that the supplier’s software is protected, freely readable (though not writable) by the supplier, and that any changes required to install the source at remote locations or any software failures are reported to the supplier in a meaningful manner (“You got it wrong, bimbo!”). Furthermore the gatekeeper will attempt to provide a service whereby redundant bug reportings and/or queries are handled by an appropriate support group so as to not interfere with the supplier’s work unnecessarily.

4.3. The Remote Site Support

Normally, little can be demanded of the site that receives a distribution. Therefore we must examine the gatekeeper’s relationship with such a site. A remote site should be asked to identify a person responsible for the remote location management of the software. Hence, the gatekeeper should attempt to foster and promote a cooperative relationship with this person. This can be done by providing easy to implement mechanisms for the reporting of problems (e.g., 4.3bsd *sendbug*(1)), a hot-line and, above all, meaningful and helpful responses to queries and problems. Effort should be made to make such reporting worthwhile to the remote site as the return will be significant.

As is the case with many aspects of this paper, the above would appear to be obvious, but unfortunately there are far too many examples to indicate that these homilies are often ignored.

5. The Database

The authors believe that formal and rigourous databases should be used to manage all the information about a software system across all phases of the software’s life cycle (i.e., from conception to retirement). Such a database and the integrated tool set (e.g., compilers that extract source from that database) would provide much better control for both the Software Engineer and Manager.

But, at this time, there are number of impediments to the use of such databases. It is currently impossible for us to employ such a database without violating some of the guidelines discussed in the previous section. The authors feel that we are still a long way from understanding how such a database would be constructed. There are concerns as to how it would be used without imposing severe constraints on the programming staff.

This section examines the D-Tree distribution information and its structuring. It is offered to illustrate the range of the required information and its uses, rather than as an example of the way it should be done. Attention is paid to the rationale for the particular representation.

The information is split into the following classifications, although the division is at times arbitrary.

- The Source − the software as a body of text and data prior to transformation to the installed form.
- Bootstrapping Information and Site Parameters − information used to establish the site and system dependent information and to initialize the tools and environment required to do the installation;
- Construction Controls − the information used to control the creation and installation of the software;

[3] There is only so far we will go.

- Distribution Management Information – data used to control, prepare, distribute and monitor the software with respect to remote sites;
- Installation Documentation – details and guidelines to aid the installer of the software;
- User Documentation – information regarding the use and purpose of the subject software.

5.1. The Source

In the large perspective, source can be considered as all the information that is required to distribute, control, install and use software. For the purpose of this discussion, ''source'' only refers to that information that is directly transformed into an installable component (e.g., the expression of a program in the C language or a data file).

The source files are stored in a directory tree. The source directory architecture strategy is difficult to describe. It is highly dependent on the relationships between components and the mechanisms used to convert the source into the installable form. However, some points must be emphasized:

- Naming conventions must be adopted that facilitates the identification of a file's type and function and the mapping of the installed file back to its sources. For example, the file that contains the ''main'' entry point has the same base name as its installed form (e.g., ''main.c'' is to be avoided).
- The directory tree should be designed to facilitate the creation of rational sub-distributions and the expression of dependencies using directory names alone.
- The directory dependency graph must be acyclic.
- Directories should be monochromatic except where it leads to severe fragmentation. By a monochromatic directory we mean that **ALL** the files of the directory are processed in the same manner.

To store past versions of the source file, there should be a versioning mechanism (e.g., SCCS or RCS). Such a mechanism must be used, in a consistent manner, for all source files that can be modified. There must be a deterministic way to map the name of any source file to the file used to administer its past versions and vice versa. It must be possible to fully install the system in the absence of the versioning mechanism.

5.2. Bootstrapping and Site Parameters

The site parameters are those system and location dependent settings that must be specified at installation time by the client. As much as is possible, such information is isolated and centralized in a well identified directory (e.g., in the D-Tree it is called *magic*). Site parameters, are specified in two configuration files (one for the machine and one for the site) that contain variable value pairs. In addition to the configuration files, the *magic* directory contains all those files that need to contain a site parameter. The first step of the installation is to copy such files to a new file, replacing the variables in the input file by the value specified in the configuration files. The directory also contains command files to install such files in the required location. In addition to the site parameters, *magic* contains the commands to boot the tools that are required to build and install the rest of the system. The *magic* directory is the most difficult to create as it must be carefully designed to present as few problems as possible. It is used before the site parameters are in place and the distribution tools are usable.

5.3. Construction Controls

By construction information, we mean the names of target directories, header and library search paths, symbolic mappings for libraries, specifications for construction tools (e.g., $CC), options, directory dependency lists, and construction specifications required to create and install the system.

In the D-Tree, two conventions are used to specify such information.

Files called **LclVars** contain simple variable value pairs to specify information that applies across a directory tree. All such named files from the root down to the current directory are processed to retrieve the settings. The more local settings override the settings closer to the root. A mechanism to select or suppress settings within the LclVars files, based on the system and options list, is provided.

The actual construction processes to be applied within a specific directory is specified via a *pmak* control file, usually called **PMC**. This file normally contains a line that specifies a shell command to be interpreted to create the full construction description file (the *pmak* script). The PMC file may also contain lines to specify exceptions and local overrides. The semantics of PMC files are described at length in [Tilbrook 86].

The directory dependency graph (used to control multi-directory constructions and to build subset distributions) is contained in a special form of PMC file. The list of libraries used by a program is specified (symbolically) within the main source file for the program. The expression of all other dependencies is created by processing the source or special rules files provided as part of the *pmak* system.

5.4. Distribution Management Information

The distribution management information may be divided into five classifications: the distribution file list (a list of all the files in a source form distribution), a validation database used to check the latter for the appearance or loss of files, a set of snapshots for remote installations (i.e., the list of files and their respective version numbers sent to a remote location), the change audit trail with checkpoints indicating releases to remote sites, and the version number database. The use and form of most of these files should be relatively obvious. It must be emphasized that the creation and protection of above information is a crucial gatekeeper responsibility.

5.5. Installation Documentation

This class of information is provided as an aid to the remote site installer and system administrator. Each source directory should contain a file called **Read_me**. These files contain information pertaining to the source files contained within the directory. For example, any special instructions or warnings are provided via this file. It should also provide a list of all the contained files plus brief descriptions, where the actual use is not obvious. Read_me files should be brief but complete as installers invariably do not read them as carefully as one might wish. The Read_me files are considered part of the source .

Three other forms of information are provided for update tapes and these are: the list of removals, the *delta* comments, and the update commentary. The list of removals names all the files that are to be removed from the previous distribution (renaming files is done by removing the old file and creating a new one). This information is automatically included by the update tape generating procedure. The *delta* comments are also automatically included in any update tape. These comments are extracted from the SCCS administration files for all *delta*s included in the update. The update commentary is normally manually produced by the distribution manager and should treated as a new source file that is delivered to the remote site.

5.6. User Documentation

Traditional UPM style manual sections (in -man format) are provided for every installed tool as part of the source tree. These manual sections are kept in either a sibling or child directory (called **man**) of the directory that contains the main component or in the directory itself. Manual section files are always suffixed by a single digit, as in *manual.2*. They should never use the suffix ‘l’ (lower case ‘L’)[4] as that as that suffix is reserved for *lex*(1) source files.

6. Version Numbers

Version numbers are very important when distributing software to a remote site. They are the key link from the programs at the remote site to the software manager’s database. As with any other database key, version numbers must be expressed in a consistent form. Version numbers must also be available in any situation where a user will be communicating with the distribution organization. Among these instances are bug reports and general questions.

In order to ensure the universal existence and consistency of version numbers, they are generated automatically by a specialized tool as part of the construction process. To meet the requirement of accessibility, version numbers are compiled into each program’s binary. The version number string has the following components:

---

[4] Unfortunately, this has become a net.rubbish common habit.

V@(#)dtzs.ms 8.1 - 98/04/01

V@(#): string used to find version string in binaries and/or core files using *what*(1).

Name: capitalized name of the program

1.2.3(4): Edition, Revision, Level and compilation numbers collectively referred to as the Version number (see below).

17 Aug 87 05:06:07: The date and time that the version string was created (usually immediately prior to compilation).

The Edition number (**1**.2.3(4)) corresponds to the edition number of the hard copy Andrew guide and will be changed by the documentation group when a new edition of the manual is to be released.

The Revision number (1.**2**.3(4)) is used to indicate major revisions or modifications to the system within the edition. It is may be incremented by the creator once the documentation is ready. It reset to 1 whenever the Edition number is incremented.

The Level number (1.2.**3**(4)) is used to indicate revisions which are discernible by the user but do not merit or require an upgrade of the documentation. The programmer may increment the level number at his/her own discretion. The Level number is reset to 1 whenever the Edition or Revision numbers are changed.

The compilation number (1.2.3(**4**)) is incremented automatically every time the program is compiled. It is reset to 1 whenever a higher level number is changed.

Each program has one of these version number strings compiled into its data section. This way, the *what*(1) program can be used to extract the version number from a binary or a core dump. It is convenient to be able to see the version number while the program is running. It should be remembered that the version number is for use by the programmer and software manager. It should not clutter up the user interface of a program.

7. Preparing a Distribution

Much of the work that goes into preparing a distribution can be characterized by the phrase “rigourous application of common sense.” The methodologies outlined here are not magical, or even necessarily original. Unfortunately, it is not clear that these methods are employed for most distributions. Therefore we outline the following four steps to preparing a system for distribution:

- Deciding which software goes into the distribution.
- Identifying the source for the chosen software.
- Laying out the distribution database.
- Establishing conventions for the distribution as a whole.

The first step in organizing a collection of software into a distribution is to figure out what is to be included. A good distribution should be easily manageable and place minimal “stresses” on the environment into which it is installed. It must make minimal assumptions about the target environment. For these reasons, one must consider very carefully what goes into the final product. Programs should be evaluated in terms of their dependencies (costs) versus their usefulness (benefits). Keeping track of many small programs can easily make the task of preparing a distribution unmanageable. It is often easier to add pieces to the distribution after the basic structure is in place and the necessary components are provided.

After the dead wood has been culled from the system, the next step is to identify the sources for the chosen components. In sizable development efforts, it is easy to lose track of where all the sources are. Unfortunately, it is not as trivial to find the correct source for a program’s as one would hope. For example, the Andrew system has been developed on a distributed filesystem with over fifteen gigabytes of storage. Within this system, there are six separate operating system distributions for four machine types, and two complete sets of Andrew software (one for the campus released software and one for the development versions). It would be for all practical purposes impossible to match the released software to source. Even within the development environment, we find things like twenty-five versions of the *install*(1) program. The process of identifying the source for programs will involve extensive tree combing. Although, largely a gatekeeper’s responsibility, this task will require a lot of software supplier co-operation.

Now that there is some idea of what constitutes the distribution, a database in which to put the source must be designed. The representation of this database will be a directory tree as described in a previous section. Its basic structure is dictated by the dependency graph of the software. The very first things to go in will be the distribution tools, which are fairly much the same from one distribution to the next. The rest of the tree will depend on the interrelationships of the many pieces of software in the distribution. Fortunately, usually there is already a running system to start from, so the process of organizing the tree can be lazy evaluated to some extent. It is not necessary at first to have all of the dependencies in the tree. It is permissible to reach out into the existing system to grab components. This gradual conversion process should make things easier for both the gatekeeper and the supplier. However, the direction should not stray from gaining a disciplined approach. The software should remain buildable during every step of the copying process, and the best way to do this is to frequently rebuild the system. Is important that development be constrained during this process. This process is complete when the entire system is organized into a source tree.

The final step is to do a global cleanup of the software system. Since the entire system is now in a "semi-organized" tree, it is possible to take a global view of the software. Certain conventions can now be established across the entire distribution. The gatekeeper should identify site parameters (i.e. pathnames where data files are found, optional configurations) and coordinate with the software supplier to establish a minimal set. All such parameters should be collected in a single directory to simplify configuration at the remote site. Conventions for portability should be established. The gatekeeper may have to suggest alternate coding conventions for use within the development environment. A trivial example of this is the use of *strchr*(3) under System V vs. *index*(3) under 4.3bsd. The distribution may provide a library which provides a *strchr* function for portability. The software supplier should be encouraged to use the portability library's version of *strchr* instead of *index*.

After the distribution has been put together, it is very important to test the construction process. It should be possible to build the entire system at a site which has never received any software from the supplier. Having a friendly beta test site where this can be done is invaluable. If possible, the distribution should be tested on all flavors of machines and operating systems it is intended to run on. Any dependencies of the software that are assumed should be carefully enumerated so that they can be announced as prerequisites for the distribution.

Usually, a distribution is prepared from an existing system. The above outlines a method of getting from the chaos of a development system to the "order" of a distributable system. Having put all this effort into creating order, it is a good goal to keep it that way. The gatekeeper will do well to try and reimpose some of these conventions on the software supplier. Hopefully this can be done in an undisturbing way. The software supplier's interest in a disciplined software approach is to increase the quality of their software. An organized source tree makes it much easier to track bugs and recognize when they are fixed. The requirement that any change to the software must be explained to the gatekeeper before it goes into the distribution will help encourage good record keeping. Historical records of the software's evolution are very useful for gaining an understanding of how the system works.

8. The Tools

This section does not describe specific tools. Rather it discusses changes to the D-Tree tools since the last paper, some approaches that have proven to be essential to manage thousands of files, and the source versioning system choice (e.g., SCCS or RCS).

8.1. The D-Tree Pmak tools

[Tilbrook 86] paid extensive attention to the tools used to support the construction and installation of software at a remote site. These issues will not be discussed in this paper, other than to state that eighteen months later the basic reasoning has remained the same: there is the need for a better approach to the system construction process.

In fact, some aspects of the approach have been considerably strengthened. The *pmak* approach to controlling construction can now be justified by stating that its use has permitted extensive simplification and/or centralization of the information required to control the construction process. This has been accomplished primarily by creating a mechanism for expressing various construction controls (e.g., include and library search paths, library mappings, target path names) in a simple generalized syntax. At the ITC, new script generators have been created or are under construction to deal with some of the new requirements imposed by the Andrew system and its components.

There has been considerable work on the dependency problems. Since one of the objectives established earlier in this paper was the simple update of a remote distribution, the mechanisms for ensuring that a generated script is accurate and up to date have been substantially improved. Soon, *pmak* will be able to infer dependencies even within a dynamically changing construction environment.

One of the objectives stated in the 1986 paper was to use whatever version of *make*(1) was provided by the host system. However, we have now reached the decision that this policy is no longer viable and imposes too many restrictions, thus future distributions will provide their own alternative to *make*. The major reason for this departure is that few *make*s can adequately or easily handle what is an absolute necessity at the ITC; the creation of objects for multiple machines from a single copy of the source. This requirement, coupled with the desire to eliminate the gymnastics required to cope with the *make* variations and limitations, has forced us to design and develop yet another variation. This seems to violate some of the principles presented earlier in this paper. But, it must be pointed out that this version strongly adheres to the KISS principle, and includes some of the key features of *mk* [Hume 87]. Also the reader should note that because this version will always be front-ended by *pmak*, it can be very simple-minded (namely, compatibility is not a concern).

8.2. E2BIG

The tool set has to be considerably enhanced to avoid the frequent occurrence of the message:

> Arguments too long.

(i.e., errno==E2BIG) when processing large file lists.

Far too many of the tools cannot cope with the file lists of even a medium sized distribution. Many of the tools (e.g., *ls*, *bm*, *grep*) have to be modified to process an input list of files as opposed to an argument list specified on the command line. Alternatively, new tools with enhanced functionality and improved efficiency may have to be created to process such lists more effectively.

8.3. The File List Creation System

In the section on the database, the major file lists were explained. One of the gatekeeper's major responsibilities is to ensure the accuracy of these lists, which will require new tools. Rather than to attempt to explain all the processing that is required, the following are the file lists that are produced during the process used to maintain the D-Tree database:

,a.all:	all files in the subject directory
,a.dirs:	all the directories in the subject directory
,c.efiles:	all the p-files with SCCS/p. removed
,c.gfiles:	all the s-files with SCCS/s. removed
,d.gfiles:	existing g-files
,f.+files:	all +files or +directory files (temporary file)
,f.commas:	all comma files or comma directory files (temporary file)
,f.else:	all files not identified by flsplit
,f.pfiles:	existing SCCS p-files
,f.sfiles:	existing SCCS s-files
,f.tmps:	temporary files identifiable by name
,m.pnos:	p-files with no s-file (s-file might have been lost)
,m.regs:	missing registered unadministered distribution files
,m.sfiles:	missing registered s-files
,m.snog:	missing g-files (s-file exists but g-file doesn't)
,m.tmps:	missing registered temporary files
,n.files:	unidentified (new) files
,n.sfiles:	unregistered (new) s-files

Normally any empty ",[nm].*" file will be removed. If any such files are not empty, they contain the names of files that will have to be processed to bring the major file lists into sync with the existing database.

8.4. SCCS vs. RCS

The current D-Tree system uses and depends on the SCCS system. For the most part this is due to historic considerations; the D-Tree has been managed using SCCS since its life on PWB (circa 1978), which was the first system to provide SCCS. However, due to the unavailability of the SCCS system on bsd systems, many people at CMU use RCS, thus we have been forced to evaluate its possible use for large scale distributions. The choice is not an easy one; choosing the lesser of two evils never is. Both have advantages and disadvantages, as outlined below.

- RCS's interface is slightly more attractive than raw SCCS, However, Eric Allman's SCCS interface is better than that offered by RCS and presents other advantages.
- RCS's ability to include the history in a source file would appear to be very useful for the distribution of small software systems but becomes costly when considering thousands of files. SCCS can be modified to provide a similar mechanism, but it is cumbersome to use.
- SCCS is buggy in some of its error condition handling and needs some fixes to deal with non-vaxen (unbelievable as this may seem) due to byte order dependent code.
- SCCS's keyword scheme makes it undesirable to ship sources with expanded keywords to external software developers, whereas RCS can deal with expanded keywords.
- RCS does not check the return values of all its writes for error codes thereby endangering the V-file. SCCS checks the return code of every write by redefining *write*(2) thus guaranteeing user notification on write failure. Unfortunately, neither system checks the uses of *close*(2), which, on the ITC VICE file system, may cause unreported failures.
- RCS does not have any checksum in the V-file thus there is no mechanism to check its integrity. Every SCCS s-file contains a checksum which is checked on every access.
- RCS writes the locking information into the V-file whenever the file is checked out with a lock which, when combined with the previous two points make the V-files very vulnerable. Even worse, from the software manager's viewpoint, the fact that the file is locked is not obvious (i.e. it requires parsing the V-file to tell if a file is locked). SCCS creates or modifies a supplementary file (a p-file) which avoids endangering the s-file and makes the fact that a file is "checked out" conspicuous.
- RCS's scheme for version management is inferior to that of SCCS for a variety of reasons. It is based on line numbers which are somewhat unreliable in the face of failure. RCS needs to do multiple editing passes of the file to build old versions, hence it is much slower. Finally, the RCS administrated source (i.e. V-file) is not expressed in a form that can be viewed directly to determine when or how a change took place, whereas that of an SCCS file can.
- RCS provides a binding scheme across multiple files, whereas SCCS does not.
- Neither system has a scheme to to deal with file removals or renaming.
- RCS always appears to have been extensively hacked to change some semantics, a good clue something is wrong, whereas SCCS is amazingly stable.
- RCS offers a merge facility. SCCS does not.

For the gatekeeper, the better choice would appear to be SCCS, partly because of the greater reliability and partly because of the use of a p-file instead of a line in the v-file to indicate work in progress. On small source systems this may not seem important, but when dealing with large systems, the ability to determine what is being edited using *find*(1) is very important. All the advantages offered by RCS (e.g., embedded history, proper management of expanded keywords, and releases) can all be dealt with in a satisfactory or better manner using supplementary tools.

At the ITC, the suppliers will probably continue to use RCS for their personal development, however, the gatekeeper will use SCCS for the above reasons and because a mature set of well integrated distribution maintenance and creation tools exist for SCCS and not for RCS.

9. Conclusions

There is a limit to the effort anyone will spend installing or upgrading a piece of software. Overstepping this limit results in user impatience and frustration, which, in turn, results in the software not being evaluated on its own merits. Software management techniques, such as those described in this paper, will substantially reduce the effort involved in installation and upgrades. Hence, these techniques are necessary for the success of any sizable project.

There are two ways in which software developed within a university research environment can be valuable. The first is through the traditional method of publication of papers in technical journals or presentation at conferences. The second is through ‘‘publication’’ of the software itself, that is making the software available in an installable form. This allows the objective evaluation of the concepts behind the project through its use. However, the costs (both in time and people) involved in creating a proper distribution are often very high. A coherent Software Management system will reduce these costs and avoid the pitfalls of installation that prevent proper evaluation of the delivered software.

The techniques described in this paper have been used effectively elsewhere. However, we are just beginning to use them at the ITC. It has been an interesting experience watching the transformation of programmer attitudes towards Software Management. The ITC software staff consists of many high quality, conscientious, and necessarily opinionated, programmers. Initially, their perception of Software Management was negative. They felt threatened and questioned its value. After three months of exposure to the principles outlined in this paper, they are very enthusiastic, and believe that it will make their job easier. Furthermore, there is a consensus throughout the ITC that Software Management will improve the overall quality of the system and make the large scale distribution of the Andrew system a reality.

Acknowledgements

Many thanks to Jennifer Robertson of the ITC for her help and patience.

References

D.M.Tilbrook and P.R.H. Place, *Tools for the Maintenance and Installation of a Large Software Distribution*, EUUG Florence Conference Proceedings, April, 1986, USENIX Atlanta Conference Proceedings, June 1986.

Andrew Hume, *Mk: a successor to make*, USENIX Phoenix Conference Proceedings, June 1987.