Managing TEX Software Development Projects

Jeffrey McArthur
ATLIS Publishing Services
8728 Colesville Road
Silver Spring, MD 29010
jmcarth@atlis.com

Abstract

During the past few years, many articles and books have been written about managing software development projects. Software development projects using TEX require some special attention. This presentation looks at the entire software development life cycle as it applies to TEX and the following issues in particular: requirements specification, design specification, coding standards, code review checklist.

Why use software development management techniques?

Writing good, versatile and well-documented code should be the goal of anyone developing macros in TEX. Unfortunately, even the macros that are included in the standard distribution of TEX fail that standard. Books typeset using TEX often have special coding scattered throughout the source to make the book layout better. The macros used, however, do not always work the way the documentation says. Book typesetting specifications are often incomplete and ambiguous or refer to the style of some other book without any detailed information on fonts, page size, and so on. Specifications can, and usually do, change and mutate during the production process. Poorly documented and bug-ridden macros make managing the process a nightmare.

Database publishing pushes the process to its limits. It is one thing to produce a book once, or even once a year; it is another thing to produce the same book each and every month, in a scenario in which the data changes on a daily basis and is extracted out of a database only for composition. Database publishing does not allow the luxury of scattering special code throughout the sources. The production process is often automated and the TEX typesetting macros must be written to take care of all contingencies because the input data file format cannot be adjusted to make the book layout better.

One solution is to use software development management techniques. This paper is an attempt to define a template or checklist suitable for managing a software development project that uses TEX as one of its primary programming languages.

Requirements specification

The first step in any software development project should be to create a document known as a requirements specification. This document defines the project in very broad terms. A copy of the requirements specification should be given to everyone involved in the project; any time a requirement changes, everyone involved should be informed of the change by receiving an updated requirements specification document. The requirements specification should include:

Description. I am amazed at how difficult it is to describe some projects. If it is not possible to write down a simple paragraph that gives a valid description of the project, that project is not well defined and has a high probability of getting out of control.

Project goal. Once the project is described a goal or goals can be defined: "If you don't know where you are going, then how can you know when you get there?" Defining the project goal can force important issues to surface at the start of the project. A written project goal provides a means to objectively measure the success or failure of a project.

Needless to say, the project goal should be documented and understood by all members of the software development team as well as the project manager. Failure to have a documented goal will lead to feature creep and project drift.

Overview of problems to be solved. Describe the fundamental problems that need to be solved. Converting the existing data into a usable format may be a real challenge. Word processing files,

poorly organized databases with little or no documentation, and spreadsheets can, and often are, the only available format for the data. All must be converted into a format suitable for typesetting with TEX.

Tasks/Functions. Specify the tasks or functions the macros perform. For example, if there are any extracted indexes or page cross references they should be defined.

Current mode of operation. It is useful to know the current mode of operation. This avoids the problem of creating a solution that cannot be easily integrated into the working environment of the user. For example, if the user is a hard-core plain TEX user, providing a set of macros that only work under LATEX would not be an appropriate solution.

Communications. Does the user expect the data back in some other format? One rather large project that I was involved in required converting data from a proprietary typesetting format into SGML, typesetting a book, and producing an electronic version of the data on CD. We finished the book and the CD and figured we were done for the quarter when the client called up and asked us about the "mag-tape version"? Our marketing department had forgotten to tell us that we had to create yet another deliverable. The data was to be delivered on an IBM formatted 6250 BPI magtape with a specific tape label. The requirements document should have specified that deliverable. Unfortunately it did not, and we had to scramble to pull together the resources to complete the project.

If the data undergoes any conversion processes it is important to specify the life cycle of the data and its changes. This means that the date (and possibly the time) when the data is frozen for production should be specified. If the data is modifed for typographic reasons during the production process, it must be specified if the original data is to be updated to match the typographic changes. For example, if the data is in SGML or XML it is common to add processing instructions to the data to help with the typography. It is sometimes necessary to make structural changes in tables, particularly CALS SGML tables,[1] to make them typeset properly. Changes of this type require complex changes to the source. If the data must be returned in a format that could be used again, the returned data must include the structural changes to the table elements.

Another example involves the creation of a printed book and an electronic product, e.g. CD-ROM. Typographic changes due to typesetting the book may need to be reflected in the CD. Failure to document this requirement can lead to serious problems if the products get out of sync with each other.

Ease of use. Specify how experienced with TEX the user of the macros should be. It is all too easy to create macros that only an expert can use. Occasionally, however, it is appropriate to create complex macros. The key is to document the expertise needed, since the level of experience of the user also defines the amount of detail that the documentation needs.

Hardware/Software. Specify the hardware platform(s), operating system(s), and implementation(s) of TEX that the macros will run on. This is important if you are using some of the modified implementations of TEX like ε-TEX, Omega, or PDFTEX.

Some implementations of TEX are compiled for a particular memory size. Other implementations are configurable. In either case, the minimum and recommended TEX memory size should be specified. This will let the user know if they have to use a larger version of TEX or reconfigure their existing version.

TEX has a very small memory footprint by today's standards. Unfortunately some of our users have just about everything running on the PC at one time. One of our users has the following programs running at all times: Excel, Word, Outlook, Internet Explorer, and several other proprietary pieces of software. "Bloatware" software packages can use up tremendous amounts of local disk space and memory. Attempt to define the possible interactions that might occur if other software packages are running. Define the amount of both network and local hard disk space required.

[1] Continuous Acquisition and Life-Cycle Support (formerly Computer-aided Acquisition and Logistics Support) (CALS) is a Department of Defense (DoD) strategy for achieving effective creation, exchange, and use of digital data for weapon systems and equipment. MIL-PRF-28001C is the military standard for CALS markup requirements. A soft copy of this is available at:

`http://www-cals.itsi.disa.mil/core/standards/28001C.pdf`

Because CALS tables are designed to support the entire DoD they are very complex and difficult to use.

Quality. Outline in broad terms the rules for word, paragraph, column, and page breaking.

Performance. TEX is a high-performance typesetting system. Usually there is no need to worry about performance. However, the generation of PDF pages on demand by a web server can become a performance issue. List all performance criteria.

Compatibility and migration. Specify if the macros are based on plain or LATEX or something else. Specify if the macros have to be compatible with any other macro package. If the macros are a replacement/upgrade of an existing package specify what amount of re-learning will be required of users as they migrate their data to the new package.

International. Specify the need to support running and/or continued heads that may include support for ® and ™ as well as accented characters. Specify how ® and ™ are to be typeset: superscript or not, serif or sans-serif or font dependent.

Itemize the languages to be supported. Each language requires it own hyphenation dictionary. The encoding of the input data should be defined. Determine if the data uses the Latin-1 encoding or some other format, e.g. UTF-8.

There is a lot of data with accented characters that uses MS-DOS code page 437 or 850.[2] This data is not compatible with the TEX standard encoding or 8r, used with most PostScript fonts. The way the data is encoded should be documented.

Service and support. Itemize the level of service and support. The days and hours when support is available should be listed in detail.

Pricing/Licensing. Define the ownership of the macros. Specify the method by which the source code will be provided and if the source code can be de-commented.

Design specification

Creating the design specification document can be done in parallel with creation of the requirements specification document, but it should not precede it. It is important to understand what is required prior to defining the design of the pages.

Publishers often provide design specifications, but publisher-supplied design specifications are often incomplete, vague, and ambiguous. Even when the publisher provides such information, a document should be created that defines all the details required by the project.

Documenting the specifications also gives the typesetter a mechanism to generate additional revenue when the publisher makes changes at the last minute.

Description. Give a detailed description of the typeset pages.

Output format. Define the output medium. Resin-coated paper and film are still used as well as electronic formats. The design specification document should unambiguously define the format of electronic files. There are many possible standard electronic formats: PostScript, PDF, or separate EPS files. If the deliverable is in PostScript, identify which PostScript level 1, 2 or 3 is required, and if the files are to be DSC-compliant (Document Structure Convention). Specify how many pages will be in each output file.[3]

If the final deliverable to the printer is to be electronic files, it is important to establish a dialogue early in the process with the printer's technical staff in order to determine the specific file formats needed.

Covers/Spines. State if book covers and/or spines are part of the deliverable.

Page size. Define the physical size of the page. In PostScript this is also the bounding box. If there are crop marks, the size of the physical page will need to be larger than the size of the trimmed page. The size of the trimmed page should also be defined.

PostScript and imposition software may have additional requirements. The bounding box of the printed page may or may not need to include the crop marks, depending on the needs of the imposition software.

Crop marks. Specify if crop marks are needed and where they are to be located. Today, many web press printers want crop marks, but they must

[2] The term "code page" refers to the keyboard and display encoding. When MS-DOS was developed there were no accepted standards for the layout of accented characters. Code page 437 was the default layout for the version of MS-DOS that what shipped to the United States, and code page 850 was the default layout for Wester Europe.

[3] Some imagesetters limit the number of pages they can process in a single file. Often long runs must be broken into ten or twenty page batches. When this happens, the design specification document should also define the file naming convention for the multiple pieces.

be located one-eighth of an inch outside the page. Specify the location of the crop marks and what they should look like.

Color separations and registration marks. Define the number of color separations and what information is printed on each separation. Registration marks should also be specified. The use of spot color or highlighting also needs to be defined.

Screens. Specify if there are any screens on the pages or if the pages are to be set on colored paper. Also define the amount that the screens must extend beyond the trim size.

Bleed tabs. Define the number, size, and placement of any bleed tabs as well as the amount that the tab should "bleed" beyond the trim size.

Makeup. Define the number of columns per page for each section of the book. The rules for starting a new column, new page, and new right-hand page should all be specified.

Fonts. Define what fonts are to be used. If the output is PostScript or PDF, the document should also specify if the fonts are to be embedded in the document. Also specify if embedded fonts must be the entire font or if the font can be a subset of only the characters used by the document.

Running heads. Define the running heads. The rules for any alpha-omega or dictionary heads should be specified. The document should also define if math, accented characters, or other complex textual material can occur in the running heads since these may require particular attention.

Even if there is initial agreement that there will be no math in the running heads, this may change later in the process. Any changes to design specification must be agreed to by all parties.

Continuation heads. Define the number and type of continuation heads. As with running heads, the document should define if math, accented characters, or other complex textual material can occur in continuation heads.

Sorting. If the data is to be sorted, the rules for sorting must be defined. Particular attention should be made to rules for ignoring leading articles such as "a", "an", and "the", casing and punctuation. The sorting order for names can be particularly complex and must be defined.

Sorting languages like Chinese can be particularly challenging since it can be sorted in either radical then stroke order or stroke then radical order. The order is dependent upon the publisher's preference. Dealing with sorting and punctuation in Chinese is particularly painful. Japanese has even more complicated sorting requirements. All of these need to be specified if the sorting involves Chinese, Japanese, or Korean.

Cross-references and hypertext links. Define the types of cross-references. The document should define if they are simple references, or actual page cross-references or even hypertext links. The formatting style for hypertext links should be defined.

Extracted indexes. Define any extracted indexes. The sorting order for each extracted index must be clearly described and all exceptions noted.

Graphics and line art. Define the parameters for line art. If there are specific limits on the art size or shape they must be documented. The acceptable formats for the artwork should be specified. If the artwork is to be scanned, the scanning resolution should be specified as well as the delivery format. If there is an approval process associated with the artwork[4] it should be clearly spelled out.

Hyphenation. Define the rules for hyphenation. Any multi-lingual document requires special attention to defining the hyphenation rules.

Widows and orphans. The rules for breaking paragraphs, columns, and pages should be defined in detail. Special attention should be devoted to pages that run short or long.

Additional breaking and grouping logic. Describe any logic that may be required for grouping. For example, it may be undesirable to break address listings except at specific places. For example: the city, state or province, and postal code should sit as a block, except when they will not fit, and then it should break following the city. The rules for addresses outside of the United States and Canada should be defined in detail.

Blank pages. The direct-to-plate technology does not come without a price: imposition software demands consistency. Often it is necessary for the final deliverable electronic pages to include blanks. When a section of a book is defined to start on a new right-hand page, the typesetting macros may need to output a blank page instead of just incrementing the folio. Bleed tabs, crop marks, and screens complicate the process.

[4] Advertisements in books are generally artwork that must be approved prior to printing in the finished book.

Front and back matter. Cover pages, copyright pages, and dedications are often created using WYSIWYG software. If the book is to be printed using direct-to-plate technology these pages may need to be integrated into the electronic files for the body of the book. How these pages are to be delivered and who will be responsible for the integration must be documented.

Typesetting deliverables. Define how the finished pages are to be delivered. In the case of film or photographic paper, include the shipping address and how the package is to be shipped. If the deliverable is electronic, specify how the data is to be transferred or shipped. Define the acceptable media: floppy, zip-disk, CD-ROM. If the files are to be electronically transmitted, list the email address or the ftp site to which the data is to be sent. If the data is to be transferred through a firewall, the security measures should be specified.

Features/Enhancements. Define possible future features or enhancements that could be made to the typesetting macros.

Exit conditions. The printer has the final word on the design specification document. The location of crop marks, bleed tabs, registration marks, and screens may need to be adjusted to meet the demands made by the printing press. Sample pages should be sent to the printer for their approval as soon as possible in the process.

Metrics

Dr. W. Edward Deming, a well-known author on management techniques and practices, introduced various quality control methods into management practice. He emphasized that you cannot manage what cannot be measured — otherwise you have no idea if what you are doing helps, harms or has no effect. He also introduced a number of statistical techniques for measuring product quality, as well as procedures for measuring improvement.

Therefore, as part of managing the development process, it is important to create an objective measure of the quality of TEX macros. As a programming language TEX has some unusual features such as the ability to change the category code of characters. This makes it difficult to create accurate metrics. The solution is to enforce coding standards.

Metrics are a complex and controversial subject that requires more than just a few paragraphs. I plan on writing a detailed paper in the near future, to cover the topic of metrics and TEX.

Coding standards

There is very little literature about coding standards for TEX. ProTeX and docstrip are tools that are supposed to aid in documentation and code generation, but neither assures consistency in coding style.

Introduction. This is an attempt to introduce a formalized set of standards for software development using TEX. During the past nine years I have managed nearly twenty man-years of extensive development in TEX and this paper is based upon that hard-earned experience. My focus has been exclusively on plain TEX, although most of these standards can be applied to LATEX.

Scope. This set of coding standards and conventions for coding and commenting TEX macros will help ensure consistency and maintainability. These standards were created not only for newly developed code; any maintenance change to existing code should attempt to bring that code into conformance with at least the commenting standards.

Purpose. Coding standards provide a framework for developing code that is both internally and externally consistent. The framework should provide the support necessary to allow the programmer to concentrate on creating the best implementation of the code.

Deviations from industry standards. *The TEXbook* defines a coding and commenting standard by its numerous examples. The coding style used in *The TEXbook* puts multiple statements per line and uses trailing `\fi`. This style makes it difficult to follow the logic of the code.

Instead of continuing to emulate the standard defined in *The TEXbook*, this is an attempt to define a new standard that treats TEX macros like any other software development language.[5]

Some will argue that the coding styles in printed books such as *The TEXbook* and *TEX: The Program* are used to save space in the printed product. Paper-saving economies that may have been exercised for budgetary reasons should not have a long-term bearing on standards particularly if the result is compromised clarity. TEX is about fine typesetting. Listings of code should be held to the same high standard as typesetting text.

There are tools to help with coding. The editor I use has a mode that is supposed to assist with the formatting of TEX. I find it more trouble than it

[5] McConnell (1993) is an excellent reference and is the basis of much of this standard.

is worth, particularly since I rarely work with the standard `\catcode` settings.

This points out the problems with tools like funnelweb and ProTeX: they place restrictions on the type of data the macros can be used with. If your input file is in SGML or XML, there is absolutely no reason to preprocess that file before using it with TEX. It is relatively simple to typeset SGML or XML data using TEX.[6]

Working directories. The TDS, TEX Directory Structure, is designed to stabilize the organization of TEX-related software packages. Unfortunately the TDS does not work in an environment where there are multiple projects that use TEX as an embedded typesetting engine or in an environment where there are multiple projects where TEX is only one of a number of tools used. It is preferable to keep all the macros under development in close proximity to the rest of the project and not part of the TDS tree.

The directory structure in use at ATLIS Publishing Services is quite different from the TDS. The top-level structure is based on accounts. Ideally, each account is broken into projects. Under each project specify the following directories:

- `doc` for Documentation and correspondence
- `help` for help files
- `version` for version control files
- `alpha` for distribution files that are part of the current alpha release
- `beta` for distribution files that are part of the current beta release
- `release` for distribution files that are part of the current production release
- `tex` for TEX files
- `sgml` for SGML, DTD files and such
- `lex` for Lex files
- `Delphi3` for Delphi 3 files
- `Delphi4` for Delphi 4 files
- and so on for other project-specific files

Format files and such are built and copied to the `alpha` directory where they are tested. When the macros have passed regression testing, the alpha files are moved into the `beta` directory as part of a 'beta' release. When the 'beta' release has been tested by the end users and accepted, the files are copied into the 'release' directory.

[6] The website `www.greenbook.net/free.asp` allows viewing of thousands of SGML documents that were typeset using TEX without the use of any preprocessor. XML is a subset of SGML.

File naming conventions. Below is a table showing the proposed file-naming conventions for macro and font files:

Prefix	Extension	Description
	`.tex`	source file
	`.sty`	macro include file
	`.fnt`	font include file
	`.dat`	data file
Tst	`.tex`	unit test file

Source file. TEX uses `.tex` as the default extension for input files. This extension should only be used for files that can be used on the command line for TEX. That is, any file which is to be run by itself through TEX or any file that can create a format file. If the file will only run with a particular format file, e.g. LATEX files, then the extension should not be `.tex`. It should be possible to determine the purpose of the file and how to process the file from its file name. Using the `.tex` extension for files that require the LATEX format is counter-intuitive because the extension implies that the file will work with TEX and does not imply that a format file is required.

Macro include file. Style, or `.sty` files, contain macro definitions. A `.sty` file should not produce any output to the `.dvi` file.

Font file. TEX provides a tremendous amount of power in its use of the `\font` primitive. Unlike some desktop packages, TEX allows complete control over how fonts are loaded and how they are used. One of the goals of good macro design is to separate form from function. That is, the data should be tagged as to its purpose and not as to how it looks. This philosophy should also be reflected in the way fonts are loaded. `\font` statements should not be mixed with macros. Fonts should be loaded as part of a separate file (or files). This makes it easier to change the fonts used to typeset a document. The document itself should not reference any font by anything but a generic name. The New Font Selection Scheme (NFSS) follows this same principle.

To promote this methodology the `.fnt` files contain all the `\font` statements. This has some significant advantages over the standard LATEX method of specifying fonts. LATEX allows the user to specify the main point size of the document in the `\documentclass` statement. Changing the main point size of the document requires the main data file be modified. It is better to separate all references to fonts and font sizes from the document.

File names. Having a portable file name versus with an understandable file name is the main

issue in choosing a standard for file names. "8 + 3" file names are more portable than long file names but it is difficult to use meaningful file names with only eight letters. All users should be polled to ensure that they can process the long file names, however. To avoid any possible problems, all file names should be limited to strictly alphabetical characters.

Coding conventions.

White space or blank lines. Blank lines should be used to show the organization of the file. Files with no blank lines are difficult to read and understand.

Dividing lines. By default TEX uses the percent sign, %, as the comment character; it also makes a good section divider. I recommend using a line of percent signs to separate sections of text. It is possible to mark off major sections by using double lines of percents. Minor sections can be delimited by using half lines or quarter lines of percent signs.[7]

Version control. Keeping track of revisions, releases, and versions of software is important to all successful software development projects. Integrating TEX with a version control system is relatively simple. There are numerous version control systems, each with their own features and syntax. Each set of TEX macros should announce to the user what version of the macros they are running. The following code fragment shows one method of passing the version information to a TEX macro:

```
% First for the revision level
\def\DefStyleVersion'#1'{%
    \gdef\StyleVersion{1.#1}}

% ***keyword-flag*** '%v (%d %t)'
\DefStyleVersion'2 (5-Aug-98 2:42:04)'
```

The version number should be announced to the user by using `\everyjob`.

General coding conventions. The Fundamental Theorem of Formatting is that good visual layout shows the logical structure of a program (McConnell, 1993:403). TEX macros should use a layout style that:

- accurately represents the logical structure of the code
- consistently represents the logical structure of the code
- improves readability
- withstands modifications as the code is maintained

Modularity. Macros should be written in a modular fashion. For example, the macro should not call out fonts by their point size but rather by their usage. So, instead of using `\tenrm` the macro package should reference something like `\NormalRoman`. This allows the fonts to be replaced conditionally depending on context.

In almost every case, over the past nine years of software development projects using TEX, the font set had to be changed at some time during the project. In many cases different outputs were created using different sets of fonts.

The same input file can be used to create dramatically different output formats. One project entailed typesetting a directory of telephone and fax numbers. The input composition file was approximately 60 MB in size. From the single composition file two different directories were created. The first, much larger, included both the phone and fax numbers. The second, much smaller, contained listings only with fax numbers. The TEX macros were written to programatically suppress listings in the directory if they did not have a fax number.

Macro coding conventions. TEX macros should be self-documenting. In other words, the code should be commented in such a way as to make it easy for the casual reader to understand what the macros are doing, no matter how complicated the actual logic is. Each macro should have a preamble comment. The preamble should define what the macro does. If the macro takes parameters, each parameter should be documented as to what it is and what its expected value should be. If the macros take optional parameters then those optional parameters should also be documented.

Indentation. To accurately and consistently represent the logical structure of the code, macros should be formatted using a block indentation style.

"if" coding conventions. All if–else–fi testing should show the logical block structure of the code. Below is an example of code that does not show the logical structure:

```
\def\strut{\relax%
        \ifmmode\copy\strutbox%
    \else\unhcopy\strutbox\fi}
```

The logical structure is much easier to understand using the following formatting style:

[7] Some editors allow the user to specify how many "repeats" of a character to use, a function which facilitates insertion of such dividing lines.

```
\def\strut{%
    \relax%
    \ifmmode%
        \copy\strutbox%
    \else
        \unhcopy\strutbox%
    \fi%
}
```

"elseif" coding conventions. TEX does not define an `\elseif` primitive. Occasionally it is useful to do several sequential tests, such as testing a parameter to see if it matches some pattern. Because the tests are sequential, a strict indentation style would be difficult to read. In those cases a modified indentation style should be used:

```
\def\TestValue{%
    \ifx\next\ValA%
        \ProcessA%
    \else\ifx\next\ValB%
        \ProcessB%
    \else\ifx\next\ValC%
        \ProcessC%
    \else%
        \ProcessOther%
    \fi\fi\fi%
}
```

Specialized TEX coding conventions. The code necessary to change the value of the TEX escape character, \, in order to process verbatim text or produce auxiliary index files, is difficult to document. In cases like this, there should be an extensive preamble that documents the process.

User interface. TEX is a batch processing typesetting system and as such has a very rudimentary user interface. However, the user should be able to tell at a glance what version of the macros they are running. TEX provides an `\everyjob` facility that allows format files to show the user the information about the version of the macros.

As TEX processes pages it displays the folio. For large jobs, it may also be desirable to inform the user what part of the document TEX is processing. This can be done using `\write` or `\message` statements.

Code review checklist

Prior to doing a code review, the software should undergo a clean build, followed by an inspection of its design and coding.

Clean build process. Prior to the release, the software should be subject to a clean build and test. Doing a clean build ensures that all the files are properly checked into the version control system and that it would be possible to recreate the project from a backup of only the source code.

Back up everything. The first step in a clean build is to back up everything. This is important because, as part of the process, many files will be deleted.

Check everything into version control. Make sure that all source files are checked back into the version control system. This also ensures that version/revision numbers are incremented.

Delete the entire project. All source code files and format files are deleted from the system. Those who are worried about disaster recovery would start with a system with a completely clean disk and would require all the development software to be reinstalled.

Restore the source from version control. Restore all source files from the version control system.

Build the project. At this point the format files should be rebuilt. One of the most common problems is missing files. If a file is missing, it was not included in the version control system.

Regression testing. Testing to make sure that software has not taken a step backward and reintroduced bugs that have been fixed previously is called regression testing. Because the entire system has been rebuilt, it is important to check that nothing has been inadvertently changed.

Design and coding inspection. The clean build should find any files not included in the version control system. Every project should have a document that defines how it is to be built and this document should be updated to list any problems that appeared during the clean build.

Using the Design Specification Document, the code should be inspected to see if it is easy to determine if the code implements the design specification. Items that should be checked include:

- page size
- crop marks
- color separations and registration marks
- screens
- bleed tab
- page makeup
- fonts
- running heads
- continuation heads
- cross-references and links
- extracted indexes
- artwork
- additional hyphenation patterns

- widow and orphan logic
- blank page generation

Project plan and status reporting

Successful management of a software development project requires that a detailed plan be developed. The plan should be created using project management software. As a rule of thumb, all tasks should be between four and sixteen hours in length. If the estimated time for task is longer than sixteen hours, the task should be broken up into sub tasks.

A status report showing the state of the project should be created weekly. Because estimating the time for a software development project using TEX is difficult, it is important that the time for the development be tracked against the plan. Tracking the time provides feedback on the estimate, allowing the estimating skills of the project manager to improve.

Summary

The Software Engineering Institute, or SEI, has defined a Capability Maturity Model, or CMM, for the software development process.[8] Briefly, the CMM levels involved are:

1. initial: no formalized procedures
2. repeatable: basic project management
3. defined: process standardization
4. managed: quantitative management
5. optimizing: continuous process improvement

This paper is an attempt to move from level 2 to level 3. The process for managing TEX software development projects must be defined so that the process can be managed, level 4, and optimized, level 5.

CMM levels provide an objective measure of the quality of the management process. Better managed projects provide higher customer satisfaction and lower costs. The standards I am proposing will encourage macro re-use and improved documentation, both of which should result in improved efficiencies, cost-containment, and easier transfer of maintenance and support duties to individuals other than the original coder.

Acknowledgements

I would like to thank the reviewer for their time and patience in reviewing this article. In particular, the introduction to the section on metrics was vastly improved by their comments.

A special thanks to Melissa Colbert and Denise Marcus, who helped me prepare this paper for submission.

I would also like to thank Christina Thiele for the thankless work as editor.

Selected bibliography

Arthur, Lowell Jay. *Improving Software Quality: An Insider's Guide to* TQM. John Wiley and Sons, New York, 1993.

Constantine, Larry L. *Constantine on Peopleware*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1995.

Humphrey, Watts S. *Managing the Software Process*. Addison-Wesley, Reading, Massachusetts, 1990.

McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, Washington, 1993.

Whitten, Neal. *Managing Software Development Projects*. John Wiley and Sons, New York, 2nd ed., 1995.

Yourdon, Edward. *Decline and Fall of the American Programmer*. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[8] SEI's website is: `http://www.sei.cmu.edu`; the CMM material begins on `/cmm/cmms/cmms.html`.