Washing Behind Your Ears: Principles of Software Hygiene

David M. Tilbrook, John McMullen

dt@qef.gts.org
Sietec Open Systems Division

ABSTRACT

This paper presents a discussion of the objectives of and impediments to software hygiene, and a list of suggestions to be followed. The suggestions are aimed at mid-sized projects, although we believe the basic ideas are applicable to all sizes of efforts.

This paper is prepared to accompany Vic Stenning's ‘‘Project Hygiene’’. We borrow heavily from that paper with respect to the basic theme of achieving better project quality through the application of some simple principles. We apply Stenning's Principles and Suggestions to source management, as well as developing our own disciplines of software hygiene.

Additional Caveats and Notes on Republication

This paper was initially given as the keynote address at the EurOpen conference in Nice, France, October, 1990. It was also given at the first Soviet Unix Users Group meeting in Moscow in the same month, but using the title: ‘‘A Dialectic on Software Autocracy and Anarchy and the History of Unix and Vodka’’. The paper has since been translated into Russian.

Since those presentations, while the principles and problems remain the same, the *qef* system has been substantially changed and improved, most notably with the addition of a graphical user interface to ease learning and project navigation. This paper has not been modified to reflect those changes except for the addition of this section and additional Biographical comments.

1. Introduction

This paper is concerned with software hygiene, which we feel is similar to personal hygiene, where software hygiene is the science concerned with maintaining healthy software. Like personal hygiene, software hygiene is most conspicuous in its absence. We do not claim that good software hygiene will help you win friends and influence people, but poor hygiene will certainly cause you to lose clients and discourage users.

Like personal hygiene, software hygiene is largely a matter of simple practices, the programming equivalent of washing behind your ears and flossing your teeth regularly. In this paper, we're going to discuss some of those practices, and (like your mother once did) try to point out some of the dire results of poor hygiene.

Our examples and our **principles** of software hygiene concentrate on medium-scale projects involving about a dozen programmers and a million lines of code, as most of our experience is in that range. However, these principles can be scaled up and down to other projects. Some examples are taken from the authors' current project, which does follow these principles. (Specific details can be found in the Appendix.)

The title of this paper was inspired by Vic Stenning's paper ‘‘Project Hygiene’’ [Stenning 89], which was to have been presented at the Software Management Workshop, April 1989 in New Orleans. Unfortunately, Vic was unable to attend, but, fortunately, both his paper and this one are being given as joint keynotes at this conference.

Stenning's paper addressed the problems of project control and management, whereas we concentrate on the control, use, and management of project source. Despite this difference, Stenning's abstract is directly applicable.

> “It is suggested that many of the difficulties encountered by systems and software projects are not the result of deep technical problems, but rather arise from a lack of basic project hygiene – from failure to enforce various elementary principles and disciplines that are self-evidently prerequisite to a successful outcome. Such disciplines are primarily concerned with the control and co-ordination of both project *activities* and project *products*.”

We will focus primarily on those activities concerned with the development, protection, application, and distribution of project product in source form. However, the disciplines we outline can be considered to be special cases of those that Stenning outlines. Stenning continues with:

> “Examination of some of the elementary principles of project hygiene suggest that the principles are often violated. However, while the failings may be obvious, the means for overcoming such failings are somewhat less so. Suggestions are made both on methods and procedures that might improve the standard of hygiene with a typical project, and on topics that demand particular attention.”

We apply Stenning’s Principles and Suggestions to the realm of source management, evolving our own disciplines of software hygiene.

> “Of course, simply focusing on hygiene will not of itself ensure project success. The successful development of computer systems and software demands a co-ordinated process incorporating effective methods and supported by effective tools. However, basic hygiene must be at the very heart of the process, and is essential to creating the conditions under which the various methods and tools can usefully be deployed.”

We will discuss some characteristics of the methods and tools that we think are necessary.

> “Ultimately, project hygiene is rather like any other form of hygiene: nothing spectacular comes from its presence, but the effects of its absence can be dramatic.”

Unfortunately in the Unix world, demonstrations of the last statement are far too common. Many software projects are bogged down in a morass of internal inconsistencies, poor documentation, versions which cannot be reproduced for testing, and frantic periods of retrofitting when it turns out that the finished product, as a whole, can not be constructed, either at the development site or (worse) at a client site.

We hold (and hope) that our software hygiene principles will help.

2. Characteristics of Software Hygiene

The characteristics of good software hygiene are relatively simple.

- **Cleanliness:** the ability to produce the product, the whole product, and nothing but the product. The procedure used to deterministically reproduce the product, in either its current form or as it existed at any time in the past, should be consistent and easily performed at short notice.
- **Consistency:** the product must be consistent with respect to its source, its installation, and the tests performed upon it. When an instance of the product is created, its behaviour must truly reflect any other instance of the product produced from the same source, modulo environmental differences (e.g., host machine and/or operating system). When the source is changed, the product is changed appropriately. The reconstructed product, incorporating the changed source, must be equivalent to the product that would be produced by a complete construction from source (a construction from the bare metal, so to speak). Further, any testing performed on the product must be relevant to the product as delivered. In other words: what has been tested is what is delivered, and what is delivered, has been tested.
- **Adaptable:** the facility with which one can transform the product to adapt to changing requirements imposed by the user and/or target environments, while preserving and protecting the consistency of the product.
- **Universally Helpful:** support is provided for all activities concerning the project not just those conventionally associated with source, construction, and configuration management.

A sure sign that some degree of good software hygiene has been achieved is **confidence**; confidence in the validity of the testing that has been performed, in the ease of correct installation, and in the consistency of the product with respect to its source. When the product is produced, one should be confident that it is what was supposed to be produced.

The above characteristics do not directly address the issues of product *correctness*. While our ultimate objective is correct software, we are actually concentrating on the consistency of the product, be it correct or incorrect. We want to ensure that problems with the product are not due to errors in the construction and/or installation process. If errors occur we want to be confident that they are due to: 1) errors in source that we can deterministically retrieve; or 2) differences between the environment on which the problem was discovered and the environment on which the product was tested and/or created. Furthermore, we want, as much as possible to be confident that if the problem is not reproducible from our source (i.e., case # 1), then it is due to a misunderstood feature of the target environment, or a problem on that environment that is not part of our product.[1]

Another reason that we appear to de-emphasize system correctness, is that we firmly believe that correctness is only possible if our listed objectives are met. Furthermore, meeting our objectives will promote and facilitate achieving system correctness.

But what is ‘‘system correctness,’’ and what other software qualities must be promoted?

In the first chapter of his book *Object-oriented Software Construction* [Meyer 88], Bertrand Meyer describes ten basic **external** quality factors, where ‘‘external’’ is defined as being detectable by users of the product. These factors are: correctness, robustness, extendibility, reusability, compatibility, efficiency, portability, verifiability, integrity, and ease of use.

This list represents a good criteria for software quality, and we claim that software hygiene comes from a concern for software quality. Consequently we consider some of these criteria from the viewpoint of software hygiene.

Extendibility Any software discipline must handle with ease the modification of existing source and the addition or removal of source. The software process used to develop, test, construct, deliver and maintain the source must be capable of producing the product as represented by the selected version of the source, no matter what modifications or changes have been made since its last production.

Reusability The software process must promote and facilitate the sharing (i.e., reusability) of software components.

Compatibility Our focus is on medium scale software projects. Typically such projects involve a great deal of integration, but their size cannot guarantee the return on investment required to employ large scale integration systems (e.g., databases, integration teams). Therefore, the development and construction must facilitate integration testing in a straight-forward and universal way, without requiring full-scale construction and testing before source can be added to the true product baseline. In other words, we want to use a cost effective way of doing integration testing.

Portability Portability is a very difficult quality to achieve. The adjective ‘‘portable’’ is often abused. ‘‘There is no such thing as portable software; there’s only software that has been ported’’ [Barry Shein, 1989]. Our experience is that writing truly portable software is impossible. All one can hope for is that the strategy one employs to cope with system differences can be quickly adapted to cope with and recover from the subtle differences that inevitably cause the system to fail.

Verifiability We are trying to ensure the validity of any verification (i.e., testing) done before the source is added to the baseline, or before the product is released. As we have said before, we want to ensure that is delivered is what was tested. However, like Meyer we want to ensure that the product can be tested once installed and that the tests are meaningful and reproducible.

[1] It is often amusing when we claim that the problem with our product is due to a bug on their (i.e., the users’) system. This is usually met with justifiable scepticism, but accepted (albeit begrudgingly) when we demonstrate that the problem does not exist on the other N (where N > 3) machines on which the system was constructed from identical sources using identical processes.

To sum up our objectives, we use the term ''rolling release engineering''. This is not to imply ''release du jour'', rather we mean that at any time, the product as it would be produced from the source, can be produced, is close to releasable quality, and, for the most part, tested.

Certainly there are many ways to achieve these objectives: Stenning concentrates on project management, with his major emphasis on project objectives and the software process. Meyer emphasizes the use of an object-oriented language (Eiffel) to avoid the pitfalls of conventional approaches to software construction. While not dismissing those issues, we are concentrating almost exclusively on activities directly relating to the maintenance and processing of source.

3. Software Maintenance — The Field of Conflict

A justification for our emphasis may be derived from the following observations from [Meyer 88][2].

> ''Often, discussions of software and software quality consider only the development phase. But the real picture is wider. The hidden part, the side of the profession which is not usually highlighted in programming courses, is maintenance. It is widely estimated that 70% of the cost of software is devoted to maintenance. No discussion of software quality can be satisfactory if it neglects this aspect.''

The 70% figure is based on traditional data processing projects, an area very foreign to the Unix community. It seems likely that the percentage is far higher for the Unix community; however, no studies have been done. Given the lack of any studies, we are forced to use studies from the D.P. developers. In a survey of 487 installations [Meyer 1988, *after* Lientz 1979], maintenance costs were broken down as follows:

Breakdown of Maintenance Costs [Lientz 1979]	
Reason for Change	% of cost
Changes in User Requirements	41.8
Changes in Data Formats	17.4
Emergency fixes	12.4
Routine Debugging	9.0
Hardware Changes	6.2
Documentation	5.5
Efficiency Improvements	4.0
Other	3.4

The above table indicates that changes constitute approximately two-thirds of the cost of maintenance. Therefore, changes represent 46% (i.e., 66% of 70%) of the cost of software. Consequently, any improvement in the process of managing changes to a product is therefore addressing close to half of the cost of software.

4. Impediments to Software Hygiene

Impediments to software hygiene can be roughly divided into three categories: those imposed by the system suppliers, by the programmers, and by the users and/or management. We had originally planned to use this section to catalogue and illustrate the multitude of sins committed by these three groups to justify some of our principles. It is very tempting to use this opportunity to subject some of the suppliers to the same kind of pain and agony that they inflict on us. But to do so would far exceed the time and space allotted to this paper, and possibly involve talking to lawyers,[3] so, we resist temptation, and limit ourselves to a short list of requests to these three groups.

To the suppliers:

- Please ensure that your documentation is read by someone other than the author before it is distributed. This might catch some of the more blatant problems, such as the directive, ''run the following four commands'' preceding a list of five.

---

[2] The authors wish to express their appreciation to Dr. Meyer for his permission to use these extracts.

[3] Or, worse yet, listening to them.

- Please read your documentation in the form that you distribute it, before you distribute it. For example, far too often ‘0’s appear instead of ‘‘\n’’.
- Please don’t distribute untested documentation for key elements. Many times it is cheaper for us to determine how your system is used through experimentation, than to recover from your outright mis-truths. For example, don’t tell us *ranlib* is required if the very opposite is true (e.g., *ranlib* does not work).
- Please ensure that **your** compilers are able to compile **your** header files.
- Please ensure that the key system development tools (*cc*, *ld*, *as*, *ar*) fail properly − report the problem and abort − when they run out of resources (e.g., disk, memory). Dumping core is not failing properly.
- Please ensure that warnings are either valid or individually suppressible. Our product construction runs processes in over 3,000 directories across six environments and produces at least 30,000 lines of diagnostics. It is extremely irritating to have to check each warning to ensure it’s not just another one of **your** bugs.
- Please ensure that compiling **your** code does not raise warnings. For example, if **your** compiler reports unused variables or unreachable code, ensure that **your** *yacc* does not generate such situations.
- Please ensure that **your** semantics are consistent within **your** own environment. We don’t really care if you conform to one of the standards, but we care a great deal about consistent behaviour within a single environment.
- Please ensure that **your** *lint* libraries describe **your** libraries. We suggest you run *lint* against the sources for your libraries and their lint libraries. This is also a way for you to test *lint* itself, so that the first time we run it, we don’t get a core dump[4].
- Please ensure that **your** *lint* is checking the same interpretation of the language definition that **your** compiler compiles.
- Please fix your C preprocessor. Whereas most of the above are requested of only some of you, this one is demanded of **all** of you, because you all have serious bugs in your *cpp*. While you are at it, why don’t you document it?

Let us assure the reader that despite appearances, the above is a very small subset of the problems. It has been limited to problems that are shared by multiple suppliers, are particularly offensive, or are serious impediments to software hygiene.

To the programmer, please realize that:

- Your responsibility is to deliver source to someone else in such a way that they can build and use the product. The fact that ‘‘it works for you’’ is irrelevant.
- If someone else has a problem with your source, then you have a problem. He or she should help to uncover the cause of the problem — it may be due to the environment or methods — but ultimately, the responsibility for rectifying it is yours, even though the original problem is not.
- You are a member of a team, and that calls for a certain minimum amount of co-operation. Certain of the coding practices you were allowed in your previous environment might be considered unsafe, unhygienic, nonsensible, or otherwise unacceptable. Please show some consideration for your fellow team-members, and refrain from such practices.
- We (the QA group) feel a lot more confident about a series of small changes than one big one. It may seem that one big change involves less work, but that is rarely reality.
- Management’s prime responsibility is to help you do your job and to help you solve your problems, but they cannot do that if you do not tell them about problems.

We may view the programmer as an impediment, but we actually believe that the majority intend to meet their responsibilities to the best of their ability. However, many are not suitably trained when hired. For most programmers, their first real experience working in a group is after they leave university. At far too many universities, Software Engineering courses are optional, very poorly taught[5], or not offered.

---

[4] Assuming that you act on your bug reports.

[5] In one introductory S.E. class, the professor announced that he had written more code than they (the stu-

Finally, to the users/managers:

- Managers will please re-read the last request to the programmers.
- Users/managers will please realize that software hygiene is a high priority and requires a suitable investment. It is not something that can be performed by a junior programmer. Furthermore, short-cuts demanded to meet short-term needs will result in long-term failures.
- Users/managers will please realize that a change in the requirements and/or priorities, requires a change in the schedules (e.g., deadlines may have to be set back accordingly).
- Users/managers will please realize that unplanned events that require any effort on the part of the developers (such as demonstrations to potential clients or visiting firemen) are effectively changes to the priorities (see previous request).
- Users/managers will please realize that programmers are (with a few exceptions) human, and as such, need appreciation and recognition for their efforts.

5. Our Credentials

A brief evaluation of the success of our approach is in order. Details of some of our software process are described in the appendix.

Our group currently consists of a project manager, a QA manager, 10 developers, and one (lonely) technical writer. Our primary project is the EMS system, which is a toolkit for building document imaging systems based on Unix, X11, and Motif. Our principle objective is to deliver that system to our German partners (Sietec in Berlin and Siemens Nixdorf in Paderborn) who will then build client-specific applications. The system is primarily targeted to Targons (SNI's RISC arch. machines), Apollos, Mips, 386s running ODT, and Siemens' MX300's and WX200's.

The deliverables consist of 19 object libraries (containing approximately 400 modules), 250 header files, 80 application programs, and a testing suite of 50 programs. The total size of the deliverable system is approximately 70 Mega-bytes, consisting of 1240 files, 700 of which are documentation related.

The second major project is the D-Tree, parts of which are used to build and maintain the EMS system[6]. The full D-Tree consists of 14 libraries (containing 200 modules) and 250 programs. The full D-Tree deliverables consist of approximately 50 Mega-bytes, consisting of 1800 files.

Combining the EMS and D-Tree systems, there are 3,500 source files containing approximately 600,000 source lines, and 13 Mega-bytes spread across 300 directories. For comparison, X11R3 is also 13 Mega-bytes, in only 1500 files.

Our approach seems to work. Both the EMS and D-Tree systems are maintained simultaneously on six environments from identical source, modulo a single construction parameterization file, yet the volume of changes to that source is huge (40,000 deltas in the last 14 months). Construction failures due to source errors are few, despite the fact that programmers are not required to test on all six environments. We have ported the EMS system to a variety of machines, with few problems, usually in hours. The problems that we have encountered are usually due to problems in the host environment (non-working *rename*(2), buggy C compilers). Most recently we ported the D-Tree and EMS systems from source to a new machine, a new operating system, and a new interpretation of the C language definition in about 15 hours (despite a painfully slow disk). This exercise revealed no problems in our system other than a number of unused static declarations (the compiler aborted when a static function was declared but not defined) and the *#define* of the C keyword ''far''. However, we did find a number of problems in the target systems: a *cpp* bug (a frequently encountered problem); libraries built with header files that were not the ones on the system; missing libraries; missing header files; disagreement between the documentation and the implementation of libc routines; system header file *#defines* without the associated support routines; and a bug in their Motif.

---

dents) would ever see. He did not announce that he also had a well-deserved reputation of never having delivered a working system. At the end of the term it was amusing to see his students desperately trying to integrate their systems for the first time, the night before they were due — a situation that this course should have been teaching them to avoid.

[6] The D-Tree system will be distributed at the EUUG Nice conference.

However, we believe that the best indication that the system is effective is that our programmers like it and value it highly. Management does not need to pressure new programmers to adapt to the approach − their peers do that for us. This might not be important to some readers, but we strongly believe that a prerequisite to a successful project is a software process that the programmers feel helps them do their job, while, at the same time, it meets management's needs.

6. Stenning's Principles and Suggestions

This section lists Stenning's Principles and Suggestions of Project Hygiene. Any accompanying explanatory text, other than extracted quotations, is to redirect (if necessary) his principle to our area of concern, software hygiene. We have limited the extracts to the minimum amount necessary to convey their intent and the reader should read Stenning's original paper, as in places we may have distorted or misrepresented his meaning.

Principle 1

Everybody involved in the project should know the objectives of what he or she is doing.

> "Or, more bluntly, everybody should know what he or she is trying to do.
>
> "This is the fundamental principle of project hygiene, and should not need stating. However, it is surprising how frequently this fundamental principle is violated, if not totally then partially."

This principle applies directly to software hygiene, without modification. In his explanatory text, Stenning refers to problems with "fuzzy" requirements or inadequate specifications, but then goes on to state

> "...The problems arise when the requirements are ill-defined but people proceed as if they were well-defined."

So too in software maintenance and development. Even if the descriptions of software components are clearly defined, misunderstandings of how, where, and when the components are to be used can lead to failures in the software construction and testing.

Principle 2

Achievement of the overall project objectives should follow immediately from achievement of all individual objectives.

> "This is the necessary complement to principle 1."

In the discussion of this principle, Stenning refers to problems of integration and the phenomenon of seemingly correct individual components failing to combine to produce a working system:

> "...[E]veryone seems to know what they are doing, everyone seems to be doing a good job, and yet the project as a whole has major problems."

This problem should be familiar to any software developer and indeed to any engineering discipline, as illustrated by the recent failures within the 1.5 billion dollar Hubble space telescope, which turned out to be caused by a software error that could have been caught by correct testing and integration procedures which were not done for budgetary reasons (sort of mega-buck wise, giga-buck foolish).

In the arena of software hygiene we want to ensure that: if a developer follows the proper procedures for the development and testing of his or her source prior to "publication", then the integration of the changed source into the baseline system will be flawless. Unfortunately, sometimes the cost of full integration testing is prohibitive (the fully installed EMS system, which is a relatively small project, requires 130 Mb of disk on a MIPS). In such cases, the programmers must convey the threat of potential problems "upwards" to ensure that in meeting their objectives they do not jeopardize those of the project.

Principle 3

Both individual objectives and overall project objectives should be realistic.

> "People tend to be over-optimistic about what they are able to do and, most especially, about how quickly they are to do it.
>
> "There is no single factor more damaging to project hygiene than unrealistic targets."

The relevance of this principle to software hygiene should be obvious. Where Stenning uses ‘‘People tend to be’’, substitute ‘‘Programmers are invariably’’, but they compensate for this characteristic with the remarkable ability to under-estimate the impact that their changes might have on the rest of the project.

One must attempt to offset the programmers’ natural enthusiasm for their product with a gentle comment such as, ‘‘We’ll get to **that** later. At this time, let’s just make sure **this** works, real [sic] soon.’’

Principle 4

There should be a known method for addressing each individual object.

> ‘‘Employing appropriate methods is an essential aspect of project hygiene — reliance on personal inspiration or hidden magic is unhygienic by definition.
>
> ‘‘When a new problem is encountered, we may well not know the method by which that problem should best be tackled. But under these circumstances we should know how we will explore the problem and determine the method by which the problem should be addressed.’’

There is an unfortunate tendency for programmers to be satisfied with a solution that works in the here and now, rather than looking for the solution which will work or can be made to work everywhere. This is particularly true when addressing problems of portability. Often discrepancies are circumvented by using ‘‘quick and dirty’’ work-a-rounds, rather than looking for a solution that ensures that the problem is dealt with everywhere and will not ‘‘break’’ any previous release of the system.

Principle 5

Changes should be controlled, visible and of known scope.

No further elucidation is required, as Stenning is directly addressing the problems of software maintenance.

Principle 6

Both people and products should be insulated from the effects of changes that are not (currently) of relevance.

It is essential that mechanisms exist that can allow individuals within a project team to postpone changes until they are prepared to make the necessary adaptation. Unfortunately, full and unlimited support of this principle will often be too expensive to implement within a medium-scale project. However, in such projects, it is usually possible to delay changes to the baseline product, while not prohibiting their development and testing. With such an approach, everyone would use the baseline software by default, but have the option to explicitly incorporate changes not yet added to that baseline.

Suggestion 1

Focus on the process as a whole, rather than on the (final) product.

> ‘‘Hygiene is not an attribute of an end product, but rather is an attribute of the process by which the product is produced. To improve project hygiene, we must focus on this process.’’

This is also true with software hygiene, which is a subset of project hygiene. If a hygienic process is used to control and change the baseline source, the production of the proper product follows as a natural result.

Later in this suggestion, Stenning states:

> ‘‘However, it should be recognised that the source code provides a very poor basis for such a process co-ordination.’’

In this document we emphasize software hygiene via a properly constituted process through which the baseline source is managed, modified, tested and used to produce the deliverables. This does not contradict the above statement. We strongly agree with it. The baseline source is nothing more than a modifiable representation of the product that can be transformed into the product using a deterministic procedure. As such, it is ‘‘a very poor basis for process co-ordination’’, but a well defined process for source management has many of the characteristics and facilities that would be required to properly support the ‘‘process co-ordination’’ of the medium scale project. Given that most Unix developments have no strategy for the management of any project information other than source, it’s a small step in the right direction.

Suggestion 2

Invest more effort in "higher level" descriptions.

> "Far too many projects still rush into code. Any time spent on higher level descriptions of the system — requirements, specifications, designs — is spent grudgingly and sparingly."

We enthusiastically adopt this suggestion, although our interpretation of "description" has to be clarified. Stenning is trying to bridge the enormous distance between

> "... [S]ome vague notion of what we want and actual running code",

whereas we are looking for better ways to represent the processes that transform the source into the product. By "better" we mean more reliable, more responsive, more flexible, more portable, more robust, more understandable, and considerably more succinct. *make*, *imake*, and even *make4*, just don't make it. They are, to what is required, what assembly code is to *miranda*[7]. This is not intended to denigrate Stu Feldman — he showed us the way, but *make* is fifteen years old and:

> "*Make* is most useful for medium-sized[8] programming projects; it does not solve the problems of maintaining multiple source versions or describing huge programs." [Feldman 78]

To describe our solution (*qef*, the D-Tree, and associated procedures) is far beyond the scope of this paper and is better done in [Tilbrook 90]. However, we discuss the essential characteristics of a software construction control system in a later section.

Suggestion 3

Co-ordinate activities as well as products.

> "As has already been discussed, there has been a traditional tendency in the software industry to focus more on products than on process, and particularly on the program code."

In some ways, we must plead "non omnia possumus omnes"[9].

Our emphasis has been directed towards those aspects of the software process that concentrate on the baseline source. Furthermore, such disciplines as Vic suggests are foreign to the prototypical Unix developer whose initial design documents usually begin with

```
main(argc, argv)
```

However, we do agree, and to that end suggest that the baseline source system provides for the organization, management, and protection of **ALL** the information associated with a project.

Suggestion 4

Adopt a strict policy on project phases.

> "Unfortunately, it is not always possible at the outset of a project to judge the realism of the objectives, simply because there is insufficient information on which to base such a judgement.
>
> "Under these circumstances, where the full scope and difficulty of the project are unknown, hygiene can be promoted by sub-dividing the project into self-contained phases — so self-contained that each phase is, in effect, a separate project."

In nearly all software projects involving the co-operative efforts of even as few as two programmers, well-defined, short-term, realistic goals and deadlines must be set, monitored, and met. The size and complexity of the individual phases must be determined keeping Stenning's 3rd Principle firmly in mind. There are programmers who can be safely assigned phases that run for months, but they are vastly out numbered by those who won't discover, or in some extreme cases, even believe, that they have a problem meeting a deadline until that deadline has past.[10]

---

[7] David Turner's (University of Kent, Canterbury) functional programming language.

[8] Historical perspective is required here to appreciate what is meant by "medium-sized": At the time the article was written, Unix could be run effectively, and not that unpleasantly, on a 250K non-separate I&D space pdp11/40 with two 5 Meg rk05s. Pure bliss was an 11/70 with a full meg, even if one had to share it with 5 other developers.

[9] (L.) we cannot all do everything — Virgil.

[10] Both the keynote speakers for this conference belong firmly in the second classification, if not the third.

Suggestion 5

Provide more semantic information to the configuration management and build systems.

It's at this suggestion that Vic drops into one of his long-term fantasies and the subject of many spirited (both figuratively and literally) Stenning/Tilbrook debates. It is also an area of great importance to software hygiene, being at the very heart of the product consistency issue.

> "As mentioned in section 2, *(the section of his paper discussing his fifth principle)* the co-ordination of continuous and concurrent changes is one of the most difficult aspects of project hygiene. While many mechanisms and tools have been developed to address various aspects of this problem, and many of these have been extremely valuable, there are still no complete solutions available.
>
> "Of necessity, in order to provide general and accessible solutions to urgent problems, most of the available change control mechanisms are based upon very simple models. The fact that something has changed may be measured, and the direct impact of that change assessed by the use of date/time stamps. Concurrent changes to individual modules are restricted by the use of check-out/check-in mechanisms. And so on[11].
>
> "Clearly such mechanisms perform a useful function, but equally clearly they address only part of the problem *(He's up in the saddle again)*. Assessing changes on the basis of data/time stamps can become extremely inconvenient when one wishes to make a limited change to some previous release of the system. Check-out mechanisms alone do not address the issues of maintaining a consistent set of modules when a change may impact several others."

Actually, this paragraph should be repeated, with emphasis. Not only are the uses of time stamps or version numbers inconvenient, there are many situations in which they are far from sufficient. In the Unix world, it is often dismaying how many people believe that running *make* ensures system consistency.

> "Over the past several years, there has been considerable progress in the fields of change management — in the context of such languages as Mesa, Modula, and Ada — from exploiting the syntactic structure of inter-component interfaces *(He's at the post)*.
>
> "There is considerable scope for further improvement by extending the checking to encompass not just syntax, but also semantics *(He's off! Unfortunately, his hobby horse is actually a somewhat pungent dead goat.)*"

Vic's motives and aims are admirable, but his insistence upon semantic checking over syntactic checking to discover process dependencies is suspect.

The primary objective of the build system must be to ensure that the product, as constructed, is that that would be produced by fully processing the selected source. Obviously, this can be accomplished by reconstructing the entire product, (provided there are no circular dependencies[12]). However, to fully reconstruct the system every time its source is changed is, just as obviously, undesirable. Consequently the use of mechanisms to eliminate redundant processing (i.e., would not produce any significant change in the produced objects) can be extremely valuable. Such mechanisms usually use some form of prerequisite dependency relationship and a model of consistency based on time/date stamp or version number.

Vic is proposing the use of semantic analysis to derive those dependency relationships and perhaps to replace the version or time stamp consistency model.

Our argument is that syntactic analysis is sufficient and, when one considers the relative costs and complexities of semantic analysis, is preferable. We base our position on the following observations drawn from our use of syntactic analysis and extrapolations of the relative cost and complexity of semantic analysis.

Every time our construction system is executed, those inter-component dependencies that can be derived syntactically[13] at run-time, are used to determine whether or not an object is "consistent". We have

---

[11] Vic is exhibiting remarkable, although somewhat uncharacteristic, restraint. In person, this sentence would be repeated at least half a dozen times.

[12] If a construction uses previously installed information, multiple constructions may be required to guarantee consistency, but such situations are due to errors in the build ordering.

[13] The "#include" of C is only one, but probably the most familiar, example of such a dependency.

worked very hard to ensure that this syntactic analysis is affordable and accurate, yet it is still extremely costly. A new implementation is being done to use a lazy algorithm (which will also improve its accuracy) and to eliminate those file system or data base queries done by the dependency generator that are also done by the consistency checker.

We realize that it is unfair of us to cite experience with X11/Motif, but, it is those packages that present the most problems[14]. We have eight such programs, excluding those we use to test X11/Motif itself. When we apply our dependency tracker against our 36 source files that use Motif or X11 interfaces, we find that the average number of header files is 50[15]. However, many of the "#includes" are required to provide declarations that are unused in the majority of compilations. Therefore, we can assume that for our typical Motif source file, the number of relevant dependencies derived semantically would be substantially lower than the number derived syntactically. An accurate measurement of the reduction is difficult, but we estimate that it would be about twenty to thirty percent. It should be noted that, if proper coding practices are followed (e.g., for every interface used within a source file, its associated header files are included) semantic analysis should not yield any new dependencies.

Through monitoring we have discovered that large majority of reconstructions are due to a change to a major component (e.g., the C source file itself), or to multiple inconsistencies (target is "out of sync" with respect to many of its header files)[16]. In either case it is unlikely that semantic analysis would eliminate any redundant processes. Therefore, this reduction would be a lot lower than the reduction in dependencies, due to the typical distribution of changes.

But at what cost?

As we have stated, our syntactic analysis is expensive, both in time and disk space. Yet, semantic analysis will incorporate the equivalent syntactic scan plus its own analysis and will need to preserve far more information. Once a change is detected, syntactic analysis can be limited to the changed file, whereas semantic analysis will require analysis of all subsequent files as a change in a file can change the semantic interpretation of a following file. Therefore semantic analysis will cost a great deal more than syntactic analysis, yet yield a marginal reduction in redundant processes.

On another issue, our system uses a relatively simple finite state automata. We support six different languages, the most complex of which has six states and fifteen productions. The addition of support for a new language is relatively simple. The creation of the F.S.A. tables is all that is required to incorporate it into the construction process. It is obvious that the creation of packages that perform semantic analyses will be a difficult and costly task, perhaps too difficult and costly to be feasible.

Finally, the problems addressed by semantic analysis are strictly within the realm of the compilation of a programming language. A large proportion of the production process is compiling, but almost as large a proportion is not. For our projects, C, yacc, and lex source files constitute only thirty percent of our total number of files. Developing an approach that can handle semantically derived dependencies, without penalizing the rest of the construction process will be difficult.

No doubt the irascible Dr. Stenning will want to respond, perhaps citing Protel — aw yes ... Protel ... management compared it to sliced bread, and the programmers hated it.

7. Dr. Tuna's Patented Software Nostrums

Up until this point, we have concentrated on generalities, commandments equivalent to "Bathing is good" and "Exercise is beneficial to your health." Now we get more specific: Dr. Tuna's Patented Software Nostrums are carefully designed to keep your system regular, your inodes clear, your backbone straight, and your disks unslipped and free of bitrot. Following these guidelines will extend your project's credibility and endear you to people who count.

---

[14] Motif could be considered to be a proof of Stenning's Law that nothing succeeds like the Second Rate, but, more likely, it's proof of Tilbrook's plagiarised Corollary that Stenning is an optimist.

[15] This is after unfolding the transitive relationships, which eliminates duplicates and repetitions. If they are not eliminated (which is the case in the compiler), it is not uncommon to find files that invoke hundreds of different include statements, since many of the required header files are repeated many times.

[16] There are pathological cases in which a single touched header file can cause the entire source tree to be recompiled, but these are usually well planned and much heralded events (sometimes done deliberately).

These guidelines are for the most part self-evident and self-explanatory; where they are not, some explanation is included.[17]

1) Give a damn! The first and foremost principle is that you must actually care about having a hygienic system. This means committing time and resources to the problem and convincing everyone involved that maintaining a clean system is important and a high priority.
2) There should one and only one source! This source should be used for **all** constructions for **all** target environments.
3) Remember that the product is the source, even if the deliverable is the result of processing that source.
4) Ensure that the deliverable product is that which is constructed from the provided source using a well-defined non-interactive single pass process that aborts if any phase fails. Minor pre- and post-construction steps are allowed, but they must be well documented and fool-proof.
5) Centralize all construction and installation parameters in one well documented and easily modified file. Procedural aids to validate the information are useful, but automatic procedures to deduce the information are difficult to maintain and frequently wrong.
6) Never require the expression of construction information twice. The second instance will be inevitably wrong.
7) Adhere to Stenning's Principles and try to follow the first four and a half of his Suggestions.
8) Make sure that everyone involved understands and appreciates the software process. Everyone in the software project must understand the mechanisms used to create, modify, construct, and maintain the product. Furthermore he or she must understand the objectives of the software process and why they are important.
9) Test, test, test, and then test again.
10) Avoid simultaneous introduction of dramatic changes. When introducing a dramatic change, always start with a known recoverable healthy system.
11) Create and frequently use a file system integrity package that checks for spurious files, missing files, obsolete files, and so on. This should check the source tree, object tree, and the installed tree. The source file system check should be done daily. One of the by-products should be mail (or news) to programmers informing them of their outstanding edits. The object tree and installed tree checks should be part of the construction process.
12) Work hard to separate the baseline, object, developers' source, and destination trees.
13) Adopt a standard source directory architecture and constantly test and tune that architecture. Construction ordering is defined in terms of directories. Watch and test for circular dependencies.
14) Limit local construction dependencies to sibling directories or fully installed sub-systems.
15) Do not restrict the methods programmers use to do their own development in their own tree. Rather, restrict the way they publish their results.
16) Test your construction frequently on as many machines as possible to catch compilation problems as soon as possible.
17) Always build the full system. You never really know the entire scope of the changes.
18) On a regular basis, do a bare metal construction — a complete reconstruction of the system on a machine that has been purged of any evidence of the system's previous existence. If the bare-metal build fails, fix the source, purge any remnants of the failed build and do it again. We do this at least monthly, and as the last test before shipping any source.
19) Ensure that the product is frequently tested against your own well-understood standards.
20) Ensure the deliverable is trivially relocatable through a change or specification of a single directory. Furthermore, with the exception of non-sharable resources, multiple versions of the deliverable should be executable in parallel on the same platform. Switching between versions should be trivial and complete (i.e., no danger of cross version talk).

[17] Further explanation can be purchased at the rate of one beer per nostrum.

21) Any faith in the standards effort is misplaced. Don't believe in them and you won't be disappointed. They are naive at best, misguiding at worst.

22) Do not insist on any standard that cannot be tested.

23) Adopt a standard for file naming and adhere to it. Overloaded suffixes are forbidden.

24) There is no paragraphing standard worth the time and energy required to impose it, other than be consistent (but reasonable).

25) Ensure that the construction system supports modification of all paths and automatic reconstruction if a change is significant.

26) Restrict baseline production installations to a small group, preferably the group that approves changes to the baseline.

27) A change to the baseline source means total re-installation ASAP and all personnel must recognize that fixing construction problems in the baseline is the highest priority.

28) Record your bugs, and make someone responsible for ensuring that they are fixed.

29) Use *lint* — when it works, it really helps. We use six different compilers, but *lint* still finds bugs that all six missed.

30) Design a test program strategy and provide facilities to quickly build programs that conform to that strategy.

31) Have others test the entire procedure of source-to-installed-client-site.

32) After any significant problems (especially after disasters), perform a post mortem. The unexamined failure is not worth having.

33) Isolate all machine dependent sections in a well known location or using a well known and consistent mechanism.

34) Centralize environment dependent aspects in single location.

35) Never use */usr/include* directly from a C source file. Always redirect through your own provided header file. This will allow you to circumvent their idiocies.

36) Never use "#include "file"". Always use "#include <file>".

37) −D to be considered harmful. If required, ensure that they are applied universally and consistently and are documented.

38) Don't overload source files by compiling them in different ways. Remember: one source file to one type of object file.

39) Develop or adopt a system whereby documentation can be incorporated within the source code files.

40) Develop and use boiler plate for often-written pieces of text, including headers and documentation.

41) Design the system to be ported, from the very start.

42) Avoid compiler-dependent tricks.

43) Avoid special case constructions. If they are useful, make them universal.

44) Avoid differing implementations to support optional semantics.

45) Use function prototypes everywhere and ensure that everywhere a function is used, its prototype is included.

46) NULL is a four letter word, the meaning of which is unclear, and whose use is dangerous.

47) Minimize your dependence upon suppliers. Expect little from them, and you will not be disappointed. Resist the urge to fix their bugs. It's unlikely your clients will have applied the same fixes.

48) Use an installation procedure that validates that **cp** did not change the size of the installed file (a not uncommon event on some file systems).

49) Use an installation procedure that refuses to install an empty file without an explicit override. This catches yet another common file system vice.

50) Ensure that the undamaged installation of every program can be trivially checked. One way to do this is to insist every program supports a flag that explains its flags and arguments (e.g., the D-Tree −x flag). Then just run every program with that flag.

51) Who cares what your programmers wear, so long as they don't shock the secretaries?

Caveat: This list is by no means complete! Each time we came back to this section, the list had grown again. Rather than attempt to offer all of the possible guides to correct software hygiene, we shall remind you of the most important nostrum of all...

Remember your two primary objectives: to have fun and make money. Any other objectives are counter-productive.

8. Characteristics of a Viable Construction System

Throughout this paper our emphasis has been on the conversion of source into the product via a well known and reliable mechanism. At our site we use *qef* (Quod Erat Faciendum) for every construction and as the driving process behind product installation. Naturally we believe *qef* is essential to good software hygiene; however, any construction system that has the following characteristics will do:

- **Constructions and Installation without Change to Source:** The construction system should support the initial installations of large collections of software, on a variety of machines, at a variety of sites, without requiring the modification of **any** source, using a trivial configuration mechanism (e.g., fill in the blanks) and a single command.

  Source is defined to be **all** the information that is created manually or must be provided by the supplier. This, by definition, includes **all** the information used to control construction and installation.

- **Termination on Error:** The construction system should halt whenever an error or condition arises that indicates that a construction has not been successful. The traditional approaches miss many failures due to inadequate checking by the tools or incorrect error returns on the part of the commands being executed. However, errors that can be rectified after a complete installation should not terminate the build but should report the problem in a consistent manner so that the correcting actions may be performed manually.

- **Porting without Pain:** The construction system should be easily ported to any reasonable Unix system and provide a totally consistent environment on any system to which it has been ported. The system should support the construction and installation of software on many different environments **without** change to the source itself. Furthermore, the system should promote portability by providing an approach whereby software can be built, tested and installed on many platforms simultaneously.

- **Ease of Distribution Upgrade or Modification:** The system should support the upgrading of a previously installed body of software by the simple addition of new source files and/or the removal or replacement of old source files, and the issuing of a single command. Furthermore, the addition or removal of source files should rarely, if ever, require the modification of the construction control files.

- **Generalized Controlling Information Specification and Use:** The system should provide mechanisms to specify and/or retrieve required control information (e.g., the target location of the installation) in or from one and only one place, or through one and only one interface. In other words, user supplied information should never have to be expressed more than once. A user should be able to feel confident that all aspects of the construction system, that need such information, should retrieve it in a manner that ensures consistency across the entire system and all uses. Furthermore, it should be possible to ensure that correct and consistent values are being used no matter what part of the system is invoked. That is to say, the information should be the same when retrieved during a full or partial construction, or the invocation of some subset of the construction system.

  An extension, or almost a prerequisite, of this objective is that the system should support the trivial addition of new pieces of construction control information. For example, if the system needs to support *whatsit* compilation, there should be a simple mechanism to specify the name of the *whatsit* processor and its normal flags through a unique and universally accessible interface.

  Finally, if the default value of a piece of control information needs to be changed or overridden within some subset of a source distribution, it must be ensured that the modification is effective over the entire subset.

- **Required User Supplied Information Reduced to Minimum Possible:** The system has to provide mechanisms to express construction specifications as succinctly as possible.

  For example, in a directory of source that is compiled and archived into an object library, it should be sufficient to state: "build and install a library called X". From this statement alone, the system should be able to generate all the required information that will perform that task.

  The mechanism used to convert simple specifications into the full construction procedures must be applicable to as wide a range of applications as possible. It should be relatively simple to add new procedures. Furthermore, it must be possible to state possible variations to a procedure easily and tersely. Such variations should not require the total restatement of the construction rules.

- **Phased Construction and Distribution Subsets:** The construction system must support partial or phased constructions such that they are absolutely equivalent to the same constructions taking place as part of a full construction. That is, the user should be able to select parts or phases of the construction to be done and be assured that the behaviour is equivalent to those same phases being done as part of a full construction.

  Furthermore, there should be few (if any) modifications required if the source to be processed is a subset of the full distribution. For example, one should not have to modify the construction control information when parts of the full distribution are excluded for economic, licensing, or other reasons.

- **Consistency Between Incremental and Full Constructions:** The system must guarantee the consistency of objects produced by an incremental construction with those that would be produced by a full reconstruction. Incremental reconstructions are defined to be those constructions that skip some construction steps if the object to be produced is determined to be consistent with its component files. The construction system should ensure that an object is reconstructed if any aspect of its construction has changed that might result in any significant change to the object itself.

- **Support for Testing and Development:** Finally, the system should facilitate testing and modification without endangering the production versions, whilst ensuring the almost trivial addition of those changes or extensions when completed. Experimentation with the source by a programmer should be economic and easy to initiate. The system should ensure that the behaviour of a test system in a non-production environment is truly reflective of the behaviour of that system in the production environment.

9. Conclusions

And thus ends our initial foray into software hygiene. We have barely scratched the surface, but the reader should now have an appreciation of the importance of a proper system construction and installation system, and the difficulties involved.

We also hope that the reader is aware that software hygiene *can* work, can be achieved, can be maintained. It can be done using the principles included in this paper. We have concentrated upon generalities, and upon techniques and objectives that can be *immediately* applied to new software projects. It is also possible (difficult, but possible) to apply these principles to existing projects.

Instead of presenting a full-grown system of software hygiene, we have tried to lay out the bare bones of a philosophy which attempts proper system construction and installation.

Anyone interested in further work in this area is recommended to think about the following topics, which have not been discussed here:

- Documentation strategies
  — styles & forms
  — administration and publishing
  — supplementary support tools

- Configuration strategies
  — natural language specific
  — {site/client} specific, options, client proprietary
  — machine/system specific

- Version strategies
  — resurrection of old releases
  — removal of files and renaming
  — documenting and controlling changes
- Testing strategies
  — Validation suites
  — Local testing, complete testing, *in situ* testing
  — Forms and contents of testing databases
- Post-release Maintenance
  — Version naming and tracking
  — Bug tracking and auditing
  — On-site trouble shooting and support
  — Upgrading and auditing of releases
- Interdependency of deliverables on other products

The objectives of applying proper software hygiene are ambitious. The task is a difficult one; any approach that attempts to solve the problems of medium-to-large scale software will itself be complicated and/or expensive. It will have to deal with a great many special cases — the ability of programmers to create new ways of complicating the lives of the software manager is unbounded.

However, we leave you with this thought from our own experience:

It's worth it.

Appendices

A. The EMS Software Process

The EMS software development process makes extensive use of the tools of the D-Tree, which have been described elsewhere [Tilbrook 86, 87, 89, 90]. A complete explanation of the D-Tree tools (such as *qef*) is impossible; however, it is the process of development that is important here.

A.1. The Source Development Process

Source which has been incorporated into the product (referred to as the baseline or ‘‘published’’ source) is maintained in a separate source directory. Each programmer works in his or her own source directory. The following is a brief description of the way in which the programmer develops, modifies, and ‘‘publishes’’ source.

To initialize his or her work space, the programmer creates a source directory. This directory is used to contain any new or modified source files. The programmer copies a prototype of the construction parameter file into this directory, edits it to set the construction-specific values (i.e., flags to compilers, destination directory) and then uses a program called *treedup* to configure the construction control file, called *TreeVars*, and to duplicate part or all of the directory hierarchy of the baseline source tree in his or her source tree. This procedure is repeated to create as many parallel object trees (trees in which the system is constructed) as are required. Usually the programmers will have a tree for their main development machine and will create object trees as required to test on other platforms.

It is important to note that the only source file used in the construction process not stored in the baseline is the *TreeVars* file. This file contains all the parameters which may differ from one construction to the next.

To construct the system from the baseline source, the programmer changes to a directory in the appropriate object tree and goes:

> *qef*

This constructs the product from the baseline source, directing any installations required to construct the system to the destination directory specified via the construction parameterization file. Note that all source files are accessed via the full pathname to the baseline source. The only files in the programmer's source tree will be files that he or she has created or has fetched for editing.

To add a new source file, the programmer simply creates the file in the appropriate directory of his or her source tree. When *qef* is rerun, new files are added to the source database that is used to create the directory's construction script. In the vast majority of cases, no further modification of the source is required to incorporate the new source file.

To modify existing source, the programmer moves to the appropriate directory in his or her personal source tree and fetches the file for editing using:

> *sc edit <files> ...*

sc was originally based on Eric Allman's *sccs* interface[18] (i.e., yet another *CS wrapper), but has been much modified to support remote source baselines. The above command fetches the named files for editing, that is, a writable copy of the file is created in the current directory, and the SCCS administration file is locked via the creation of a p-file. The existence and visibility of this p-file is an important factor in choosing SCCS over RCS. When one is administering 3500 files over 300 directories, a quick way to find all the files currently being edited is crucial. We have also modified the p-file creation routine to embed the name of the host and directory of the *fetched* file in the p-file. We also strongly believe in ensuring that files are not being modified simultaneously by multiple programmers. RCS supports this as an option that can be easily circumvented or over-looked.[19]

The programmer then edits the source as required. When *qef* is rerun in the object tree, files in the programmer's source tree versions of the file take precedence over those in the baseline.

The procedure of fetching files for editing into the programmer's own source tree and rebuilding is repeated as many times as is required to implement the current development.

Note that throughout the development, the programmer's source tree need only contain those files required that are either new or modified versions of baseline source and the object tree contains only files that are constructed. This total separation of baseline source, modified source and constructed files has many advantages. Perhaps the most popular among our programmers are: only the command ''ls'' in the source directory is needed to determine what files a programmer has changed; only ''rm *'' in the object tree is needed to remove constructed files.

When the programmer is satisfied that the modifications are correct and they have been tested (preferably on multiple platforms), the programmer **push**es the modifications using:

> *sc push files ...*

These commands ''push'' the named files to **unpub** (unpublished source) directory, which is used as a quality assurance administered buffer between the programmer and the baseline source. The programmer is required to provide a description and rational for the changes to the file via a specially named file, a −y flag (a la *delta*(1)) or as input.

At regular intervals (usually daily), the QA group checks the *unpub* directory for unpublished files. Time permitting[20], the product should be constructed from the *unpub* source (it's just another source tree) and tested before the modified source is added to the baseline. Given that the change is acceptable, the QA group ''publishes'' the modification. This *delta*s the changes into the file's SCCS administration file, updates the baseline g-file (the source in clear test), and adds an activity record to the delta audit trail. It is then QA's responsibility to construct the baseline product on all the machines on which there is on-going development (machines used for demos and full release testing are excluded). If any problems are encountered, they are immediately rectified. If a programmer introduced an error in the construction due to one of his or her modifications, it is of the highest priority to rectify the problem immediately and convey necessary changes to the QA group.

---

[18] We chose SCCS over RCS for a variety of reasons, most of which were outlined in [Tilbrook 87] although some are mentioned in this paper.

[19] There is an work-station supplier that prides itself on providing CASE tools that support simultaneous modifications of source files. We have one of their products and it's the least reliable system in our office. The inference is clear.

[20] It never is.

A.2. EMS Documentation

The EMS system's primary objective is to provide a tool-kit with which others may build client-specific applications. Consequently, documentation of the basic system and, in particular, of the 700 subroutines in the provided libraries is of utmost importance.

In our approach, documentation is treated in exactly the same manner as any other source. It is maintained and modified using the same tools and procedures. In fact the mechanism to "produce" the documentation is the same as to produce the other products (all one ever says is "qef").

To document subroutines and libraries we use database entries (referred to as *man3db* entries) embedded in the source files. The *man3db* program extracts these entries and creates a variety of products including: the traditional *nroff* man file; an indexed fast retrieval database; a software inventory; *lint* libraries; and a prototype database. One important aspect of this approach is that the documentation is *lint*able, in that the synopsis as defined in the *man3db* entry is checked against the function itself using *lint*. Most importantly, any time a programmer is modifying source, he or she is also modifying the file that contains the appropriate user documentation. Consequently, changing the documentation to agree with changes in the routine's semantics and/or interface is simply a matter of scrolling up a page in the editor.

Finally, all the documentation associated with the project is stored in the project source tree. This includes the project plan, designs, specifications, discussions of those descriptions, installation notes, testing reports, project inventories, validation suites, post installation instructions, the bug database, change audit trails and information to resurrect previous releases.

A.3. Bug Tracking and Testing

Every source component in the EMS source tree is assigned an owner — the developer responsible for the correctness of the component[21]. Every component that is to be released, is assigned one or more testers (developers not involved in its maintenance). A tester is responsible for reviewing the code for completeness, conformance to the few coding standards that exist (see nostrums), agreement of the component's code and its documentation, the completeness of the documentation, and for the creation of the component's testing procedures and tools. Testers must review and test those components which have been assigned to them, though they may report problems wherever they find them.

Whenever a tester discovers a problem, a bug report is prepared and mailed to the QA group. The bug report contains the source module that contains the error (if known), the priority of the bug (p0 for "cosmetic," p4 for "renders system unusable"), a one-line description of the bug, a long description of the bug (including how to duplicate the problem), suggestions as to how the problem may be rectified, the environment in which the bug occurs, the date the bug was reported, and the reporter's name.

On receiving a bug report, the QA group attempts to validate it and checks if the bug has already been reported. If the bug is legitimate and new, the bug report is assigned a reference key and the owner of the source file is assigned the bug. The bug report is added to the bug database, mailed to the bug's reporter and the module's owner, and posted to the bug-newsgroup.

The owner is then responsible for fixing the bug, testing the fix (if appropriate), "pushing" the fix, informing QA that the fix has been pushed. QA then asks the bug reporter to verify that the fix is indeed correct and has not introduced new problems.

To encourage this process, the team is awarded minor bonuses for reaching plateaus of fixed bugs (most recently a beer and pizza lunch). On a daily basis, bug reporter/owner counts, graphs and tables of the bugs, and the countdown to the next free lunch are produced and posted on bulletin boards.

It must be noted that performance reviews are not based on bug counts. To do so would require more rigorous evaluation of bugs than they merit or than can be afforded and would discourage the reporting of minor problems and inconsistencies. We foster mild friendly competition but avoid reprimands. The overall emphasis is that this exercise is a team effort whose objective is to improve the quality of the software to the entire team's benefit and satisfaction.

---

[21] Some organizations (e.g., UCB's CSRG) use the scheme that the last person who modified the file is responsible for it. However, this scheme discourages people from fixing cosmetic problems or minor errors that do not require a complete understanding of the module.

Bibliography

Feldman 78 — S.I. Feldman, *Make − A Program for Maintaining Computer Programs*, Unix Programmer's Manual, volume 2A, Seventh Edition, January, 1979; Bell Laboratories, N.J.

Lientz 79 — B.P. Lientz and E.B. Swanson, *Software Maintenance: A User/Management Tug of War*, Data Management, pp. 26-30, Apr. 1979.

Meyer 88 — Bertrand Meyer, *Object-oriented Software Construction*, Prentice-Hall, 1988.

Stenning 89 — Vic Stenning, *Project Hygiene*, Usenix Software Management Workshop, New Orleans, 1989.

Tilbrook 86 — 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.

Tilbrook 87 — D.M.Tilbrook and Z.Stern, *Cleaning Up UNIX Source -or- Bringing Discipline to Anarchy.*, EUUG Dublin Conference Proceedings, September 1987.

Tilbrook 88 — David Tilbrook, Dean Thompson, Mark Lorence *A New Look at Some Old Problems*, Unix Review, April, 1988.

Tilbrook 89 — David Tilbrook, *Under Ten Flags (not always smooth sailing)* Usenix Software Management Workshop, New Orleans, 1989.

Tilbrook 90 — David Tilbrook, *Quod Erat/Est/Erit Faciendum*, The EUUG Fall 1990 Distribution Tape.

Biographies

David Tilbrook received his B.Sc and M.Sc. from the University of Toronto in 1974 and 1976 respectively. His masters thesis was on NewsWhole, an interactive newspaper pagination system, and the first graphics system to use iconic cursors. From 1970 through 1978 he served as a consultant to the Metropolitan Toronto Library Board on cataloguing and computing systems. He was a co-founder of HCR in 1975 where he stayed until 1978 when he joined Bell-Northern Software Research as Senior Member of Technical Staff. At BNSR he worked on MASCOT (a design and construction approach for real-time systems), office systems and information technologies. In 1981 he joined Systems Designers Ltd. to continue work on MASCOT with Ken Jackson (co-inventor of MASCOT). At SDL he also worked with Vic Stenning on a variety of small projects, and in 1983 David joined Vic at Imperial Software Technology in London. He was an associate director of the Information Technology Center (ITC) at Carnegie Mellon University (CMU) from 1987 to 1989, where he tried (and failed) to bring order to anarchy. David has now returned to Toronto to work for Sietec OSD. They think that he is the Quality Assurance Manager, but he is actually the Software Hygiene Research Group. David is an honourary life-time member of the EUUG, for whom he was conference chair three times. This is his 10th presentation to an EUUG conference.

John H. McMullen is acutely aware that his experience is dwarfed by that of his co-author (but then, so is his body). He received his B.Sc. from Waterloo in 1984 and spent several years writing user's manuals for several companies using a Different Operating System. John is now in Toronto working for Sietec OSD as a Technical Writer and hopes someday to improve David's spelling. This is his first paper presented to an EUUG conference.

Since the original publication of this paper in 1990, both David and John have left Sietec, although David continues to have a close relationship with Sietec, most recently to port the D-Tree to Windows/NT. David is now the president of **qef** Advanced Software Inc., a company formed to create, market and sell the **qef** product, in partnership with a Californian distributer. John is working for Softway Inc., in Waterloo.