An Unorthodox Approach to Undergraduate Software Engineering Instruction

Robert A. Morris
The University of Massachusetts at Boston

ABSTRACT: Software engineering principles can be taught to inexperienced undergraduates by substituting code reading, maintenance, and enhancement for the more usual beginning-to-end team project. The study of mail reading systems of intermediate size proves a suitable environment for the study of complex systems.

1. Introduction

Traditionally, software engineering teaches the skills which are widely regarded in professional circles as enhancing software productivity. These include study of the software life cycle, principles of specification, design, implementation and maintenance, rapid prototyping, team programming, and documentation. The approach in such courses is almost always to divide the class into

This work supported in part by NSF Grant IRI 87-15960.

teams to work on a major piece of software from conception through implementation. Sometimes the teams compete, sometimes they cooperate, and sometimes they work on altogether different projects, depending on the resources available and size of the class. Perhaps due to the influence of the ACM Curriculum 78 [Austing et al. 1979], which suggests such an approach for a course in Software Design and Development, virtually every recent meeting of the ACM Special Interest Group on Computer Science Education contains papers on experiences in such a course (see, for example, Bullard 1988, Carver 1985, and Collofello 1985).

This paper describes an unorthodox way of teaching undergraduate software engineering which the author utilized during the Spring semester, 1988 at the University of Massachusetts at Boston (UMB).

The standard course at UMB is similar to the Curriculum 78 course mentioned above. Although it does not have substantially more pre-requisites than we describe below, it seemed to the author that the traditional approach depends for its success on the students already being skilled programmers and, in particular, already having some exposure to large software systems. At UMB, such exposure is offered in several elective courses which are widely regarded as demanding. These include Operating Systems, Artificial Intelligence, and Graphics, in each of which there is substantial focus on system issues. Each course makes heavy demands on the programming skills of the students and those who complete the courses usually emerge from them quite strong. However, a number of students are not able to survive in those courses, which accelerate quite quickly and assume the students are already reasonably proficient in the relevant language (usually C or Lisp, as appropriate). It was felt that students whose background or ability excludes them from those courses nevertheless need a course which strengthens their skills in design and coding.

The target students also have little exposure either to big systems or to system development tools such as source level debuggers, cross reference programs, or system compilation tools such as the UNIX *make* utility. Thus, to this author, the relevant question seemed to be what approach would offer students without the traditional expected background a course different from a low level "UNIX systems programming course," which so

often reduces just to practice using utilities. An answer was to focus on an existing large system which the students would debug, modify, and enhance. This also was deemed to be somewhat realistic, in that new professionals are typically assigned tasks in code maintenance, if only to familiarize them with the system at hand.

Although study of an existing large system does not give a "beginning-to-end" experience, it allows the possibility of showing real accomplishment in a short time. This might not happen if inexperienced students are thrown into the design waters to sink or swim, even with the aid of the instructor. One drawback is that it becomes more difficult to discuss design issues in a global way, since the design is already in place. On the other hand, discussion can be focused on the success or failure of the design of the studied system, especially with regard to its robustness in the face of enhancement requirements.

Code reading as a method of study is not novel (see Deimel 1985 for the arguments, an extensive bibliography, and an appendix on the subjective measurement of program reading comprehension). Our approach certainly is consistent with a program reading approach, since reading code is manifestly the principal tool in program enhancement. However, we go well beyond that to include design and coding of new capability, an approach that does not appear to be very common. A survey of undergraduate Software Engineering courses [Mynatt 1987] found that 40% of such courses gave only limited attention to maintenance and modification, 18% gave none at all, and only 12% reported in-depth attention to these issues.

2. *Choice of the System to Study*

After careful consideration of some alternatives, including editors, compilers, CASE tools, and window systems, a decision was made to study electronic mail systems. Electronic mail is a rather broad topic, and several components of mail systems can be distinguished, although, in practice, some components may be implemented in a single program. We can divide mail tasks by function: writing, transport, and reading. In many environments the writing and reading functions are executed by a single program

which contains facilities for editing outgoing messages, filing incoming messages, automatically constructing reply addresses in response to incoming mail, and handling similar matters of convenience. A transport agent, however, may be thought of as having the responsibility of accepting mail from writers or other transport agents and delivering it in a form obtainable by readers, passing it to other transport agents, or arranging for notification of its inability to dispose of a message.

The functionality described above is exemplified in the UMB Computing Laboratory, which has a fairly typical local network based mostly on the 4BSD Berkeley versions of UNIX running on Sun and VAX systems. As in most such networks, there are a number of reader/writer programs available, but a single locus of transport on each host, the *sendmail* program, which is invoked by writers but never by users in normal circumstances. For the final delivery of mail to a user's mailbox, the *sendmail* program on the addressed host will invoke a somewhat primitive integrated reader/writer/transport program to write the message. This program, known simply as *mail* is the original and only mail facility provided in early versions of UNIX, although a more sophisticated version, *Mail*, is more often used as the reader/writer in Berkeley UNIX systems [Shoens 1986]. Although not normally invoked by users, an undocumented option to *mail* permits the direct delivery of mail to the addressee's mailbox, and it is this facility which *sendmail* uses. Under normal use, *mail* itself also requests transport service of *sendmail*. In addition *sendmail*, operating as a daemon, exchanges mail with *sendmail* programs on other hosts on the network and with other transport agents, notably agents for dealing with Internet mail via a telephone connection to a CSNET relay machine, and dealing with UNIX *uucp* mail on telephone and serial line connections. (For a brief description of Internet domain-based addressing and of *uucp* mail, see Quarterman & Hoskins 1986. That article contains an extensive bibliography and gives substantial insight into the problems of the transport of mail and other data across networks.) The programs visible to users are integrated reader/writers (we will call them simply "readers"), ranging from simple line-oriented programs to screen-oriented software invoking (or sometimes invoked under) sophisticated editors.

In trying to narrow the focus of study, it became clear that studying mail transport agents would present a consequential difficulty in an environment not devoted entirely to the developers: installing and testing modified software can not be done without impact on the other users. (Indeed, debugging code in *sendmail* provided an entry point for the infamous Internet virus in November 1988.) Mail readers, however, have several advantages not all enjoyed by some of the other systems considered.

First, every student would already have had extensive personal experience as a user of mail systems. The pre-requisites to the course essentially imply that the students would be junior or senior Computer Science majors, with at least 4 semesters of programming in Pascal, a data structures course, and a year-long lab course in low level computer system architecture. Some would have had a Scheme-based course in Programming Languages in which they would have studied the structure of programs but not large systems. They would have had extensive experience with at least two mail systems, typically VAX VMS and UNIX mailers. Thus, there would be a common body of experience to serve as a focus for criticism of system design.

Second, a large variety of mail readers would be available on our UNIX systems, almost all of them in source form and all with extensive documentation. This would make the comparison of systems feasible, especially as to user interface and feature sets. Of course, since source code is available, enhancements would be limited only by the design of the target system. This limitation itself was an area of study in the course.

A third advantage would be that mail readers are not self-contained. Since they must interact with a wide range of other software, from editors to mail transport systems, opportunities would arise to discuss external interfaces and even, in some cases, international standards. This benefit also applies to mail transport systems, but modifications to the local mail transport would impact all system users, not just the developers, and so would not be easily tested in an environment shared with other users. Of particular importance is that messages constructed using any enhancements to a given system not break other systems.

A fourth benefit emerged only after the course was underway – one which may be unique to mail systems – namely, that there is

a non-computer version of the system, the global paper mail system, which is familiar to all and which can serve as a sounding board for design questions. For better or worse, most existing electronic mail systems are easily modeled by the paper mail system; it shares with electronic mail properties of local variation combined with the requirement that local variants be able to exchange messages. In one course exercise, the addition to the mail reader of a "return receipt" facility, examining the differences and similarities between electronic and paper mail proved particularly useful for design and requirement specification.

3. *The Projects*

The *Mail* program provided with the Berkeley versions of UNIX was selected as the reader to study. Because the students were unfamiliar with development tools, the course began with a bug deliberately introduced into *Mail*. *Mail* determines whether text in the user's system mailbox is a legitimate mail message by parsing the date field in the first line of its header. If all the dates in the system mail box are invalid, *Mail* will assert that there is no mail for the user.

The date parser in *Mail* was modified so that it always required a two digit date, instead of two or, optionally a blank followed by one digit. Seeing only its behavior, and given access to the entire, buggy, source code set, the students were expected to find and correct the bug. By a happy coincidence of schedule, the assignment was given on the 1st of the month and could then be made due one week later, on the 8th. This gave the students one day of grace before the bug would lie fallow for another 20 days, undetectable until so far beyond the assignment due date that no credit would be given!

This bug has several pedagogical advantages typical of real bugs and worthy of study. First, its manifestation, the fraudulent "no mail" message, appears in the thread of control at a fairly great distance from the erroneous code. This means that some tracing of the logic is necessary. Second, it is easily ascertained to be a bug. This means that no time need be spent capturing

instances of the bug, a subtle and difficult problem for many real world bugs but beyond the ability of students whose first exposure this is to source code debuggers. Third, the bug manifests itself on mail produced both with the modified *Mail* and with all other mail producers, both on and off the local network the students use. Since other mail writers can induce the bug, this suggests that the problem can be isolated to the received mail processing parts of *Mail*, potentially limiting the scope of debugging. A final "advantage" tugged too much at the teaching sensibilities (or perhaps the sympathies) of the author to be allowed to stand: some conditional debugging code lay commented out of the distributed sources. This code, when turned on, should lead the bug tracker more quickly to the relevant source file. However, when it was turned on, the code proved not to signal the bug! In fact, the code could never have worked as distributed because it was testing a boolean variable which was never set, and the students were given code with this corrected, so that the bug manifestation then gave them substantial information, namely that the header was faulty.

The main aim of this exercise was to expose the students to elementary software tools such as source debuggers (they were required to use *gdb*, the C source debugger from the Free Software Foundation [Stallman 1987(1)]), cross reference tools and tools for finding occurrences of particular identifiers or strings, the UNIX *make* utility for rebuilding systems after changes, etc. The usefulness of these tools apparently was evident to the students, as they continued to use them throughout the course without further prompting. The exercise also familiarized the students with the *Mail* source set, which comprises about 10,000 lines in 27 files.

The next exercise in the course was to add a capability to the (fixed!) *Mail* program. It was decided to add a "return receipt" facility which would function much like that provided by paper postal services: if the sender requests such a receipt, the reader is not permitted to read the mail without generating a receipt.

Two realities became apparent in settling the specification with the class. Both provoked worthwhile discussion, as well as an introduction to the nature of Internet mail and security. The first was that, in most UNIX systems, file system access privilege is a linchpin (or, one might say, "lynchpin") of system security and

almost always is delicate and obtrusive. In the case of *Mail*, each user's system mail box is readable by that user, although by no one else except privileged processes. This means that any requirement that mail be unreadable without generating a receipt can be easily circumvented, for example, by simply invoking an editor on the mailbox. Many mail systems have this kind of resource readable only by the mail system itself, but making such a change on our network would be potentially disruptive to other users, since shaking out the consequence of permission changes in UNIX systems typically is a time consuming and often frustrating task. The second reality was that, no matter how security issues were resolved locally, every mail reading system (of which there are at least 6 on our own network), including those on any host to which mail could be sent globally, would have to obey the same requirements.

In the face of these security issues, it was decided to have the students solve the problem only for the Berkeley *Mail* program. This was thus an exercise in exploring what the design, user interface, and implementation issues might be. Since the security issues probably do not have an appropriate solution, the exercise is somewhat academic, but this did not seem objectionable in view of the fact that the problem was easy to state and so presented a suitable vehicle for discussion of the nature of specification.

Initially, no guidance was offered students about what is a specification. They were given a week in which they were required to discuss the problem with each other personally and online. All of the online discussion was recorded in a bulletin board facility available to all the students and especially suited to dialogue. At the first subsequent class, the students were quite surprised at the author's observation that they had thoroughly mingled implementation, user interface, and specification issues in their dialogue. They came to the class expecting to settle a complicated specification and left after hammering out a three line definition of how a Return Receipt Requested (RRR) object is to be recognized and disposed of. The class decided that RRR mail should have a header field named X-Return-Receipt-Requested, conforming to the RFC822 [Crocker 1982] recommendation for the addition of "unofficial" header fields in Internet mail, and that

a mail reading program should not display the message, nor include it in any other message, without generating a receipt. Further, a subsequent invocation of the same program should not defeat the specification.

By intent (of the author), the simple specification did not address any user interface issues, which were to be discussed later. The specification could be met by a program which silently generated receipts for all RRR mail or, indeed, for all mail whatsoever. It had the virtue, however, of separating user interface from object specification design. This, the author contended to the class, is a goal whose successful pursuit can make software more flexible and longer lived. The next week was spent specifying the user visible behavior of RRR mail and implementation issues. The author acted as moderator in the online and class discussion, attempting to keep distinguishable issues separate from one another. About a dozen issues were isolated, ranging from the nature of message identifiers to questions of preventing "RRR loops" in which automatic receipts to RRR mail were themselves generated as RRR mail.

It is interesting to note that the Return Receipt of the U.S. Postal Service is not mail from the original target, but is actually mail from the postal service itself, which addresses certain kinds of loop, security, and authentication issues. It is also interesting to note that the U.S. Postal Service will accept Return Receipt Requested mail for any domestic addressee, but only mail destined for particular foreign countries can get such service. This corresponds to the fact that other electronic mail services need not recognize unofficial header fields and still remain in compliance with RFC822. In all, the narrow focus on the RRR facility proved to be an excellent forum in miniature for many of the same issues which would arise in designing a complete system, and the Postal Service provision of this capability had a number of properties worthy of examination.

In part due to the author's interests, and in part because so much of the RRR design centered on user interface issues, the final project in the course explicitly centered on user interface design with particular emphasis on the effect on the user interface of other design choices in the system. Several weeks were spent examining, comparing and critiquing other mail readers. These

included two widely available screen-oriented readers – ELM [Taylor 1986] and the Emacs mail reader RMAIL [Stallman 1987(2)], and MH, an alternative mailer circulated with the Berkeley UNIX distribution [Rose & Romine 1984]. No code was examined, but there was some discussion of the impact of these systems' behavior on projects such as the RRR one. For example, RMAIL permits the editing of all fields in the message header, including the message id field, and makes mail forgery substantially easier to commit than do other mail readers; ELM has a "bounce" facility which permits the mail reader to appear as part of the transport system, forwarding mail with little indication that there was intermediate examination.

The final project for students was to attempt to add an X-windows Systems interface to *Mail*. A similar approach is taken in the Mailtool provided on Suns for the obsolescent SunView [Sun Microsystems 1986, Ch. 6], and also in *xmh*, an X-windows version of MH in the contributed software in the X11 distribution. There were two goals in this assignment. First, few of the students had exposure to window systems and none had exposure to window system programming. The X paradigm of the client-server model was entirely new to all the students and none had done any event-based programming. In addition, although not strictly object oriented, the C library interface to X [Scheifler et al. 1988] is clearly influenced by object-oriented programming. It was decided to use this interface instead of the higher level toolkit distributed with X11 mostly because fewer abstractions needed to be discussed. Although this decision warrants more consideration, the author's present opinion is that simple low-level programming is an appropriate prelude to the more productive toolkit approach. It may also reveal more of the generalities of window systems. The second goal was to examine the extent to which the radically different X model of I/O could be imposed on the UNIX stream oriented "standard I/O" package heavily embedded in *Mail* in a way that involved minimum disruption to the *Mail* internals.

The students were first given a small project to familiarize themselves with X programming. This was a file reading program analogous to screen-based readers that pause after a screenful of text and give the user a chance to read the text before continuing at the user's request. Most of the subtle X issues can be dealt with

by borrowing code from the excellent example distributed with X11 entitled "Hello world" [Rosenthal 1988].

The major problem was to design and implement an X interface to *Mail*. Since only about three weeks remained in the course, it was left as an option to do mouse-based input. Only the very strongest students attempted mouse-based command processing, but it is in fact not very deep, at least for argument-less *Mail* commands, since windows are a cheap resource in X. For this reason, it is straightforward to associate with each mail command a "button" comprising a separate window. When a mouse button is clicked on such a software button, the system executes the command as though ordinary keyboard input were given to *Mail*. However, the specification on which the class settled required only a separation of output based on semantics: mail headers were to go in one window, message text in another, system interaction in a third, and a fourth was to contain a help facility consisting simply of a list of the commands available to the user. *Mail* provides a brief keyboard command interface, and the simple implementation merely took this input in the system window.

The fundamental issue to be addressed is when to intercept *Mail*'s output to direct it to the appropriate window. The interesting part of this problem, and a similar one at the input side, is that *Mail* is a sublime example of UNIX stream-based I/O. On the one hand, good engineering practices have insured that all of the I/O is funneled through a very few routines. On the other hand, by the time those routines are invoked, it is difficult to determine whether the output is destined for a terminal, a file, or another program. This is, arguably, not a useful paradigm in simple window programming, where windows are radically different entities from other things modeled by streams.

Since the aim is to interfere with *Mail* source as little possible, decisions have to be made about how far into the thread of control (or, equivalently, in what generality) the output should be intercepted and re-targeted to window system requests. This problem is complicated (in rather realistic ways) by some intrusion in naming of the original programmer's unstated model of *Mail*'s tasks. For example, one important routine named *send()* might reasonably be assumed to deal with sending mail messages. Actually, it is concerned with sending text to various streams,

which makes it much more important in our context than its name might suggest (recall that the goal is to avoid interference with components that actually deal with mail, and only to modify I/O). In other cases, there are certain tasks which one might expect to be dealt with at a logical level sufficiently high that they are not encountered in this problem. But some of these proved to be scattered at various levels of the thread of control, sometimes necessitating special case treatment where one not would expect to confront these tasks at all. One such instance had to do with the output treatment of parts of the header information, which, because it is so voluminous, can be selectively suppressed by the user. In some cases the "kludges" we stumbled on seemed to have the scent of afterthought, which presented a good forum for discussion of how to write code that survives addition of functionality.

With the author's encouragement, there was substantial mutual help even though there were no formal teams. As in any course, this brings the risk of the strong students carrying the weak, but in the current instance the enrollment was small enough and there was a highly qualified and experienced graduate student assistant, so that a close eye could be kept on the students who were struggling.

Finally, it should be mentioned that, besides the programming focus, there was a series of readings on related topics (see Reading List). The readings were to provide the students evidence that they were not alone in facing the issues we dealt with in class. Except for scattered random discussion, no class time was devoted to this, but a final exam attempted to measure whether the students had extracted from the papers the elements common to their own experiences in the course and generally they had. Also, two guest speakers from industry addressed the class on large systems – one about the building of a large expert system and the other about portability issues revealed in a large document production system.

4. Summary and Evaluation

The principal omissions from a traditional course are the absence of team programming and the absence of a start-to-finish system building experience. Also, the utility of rapid prototying to test design is fundamentally absent in a setting where the design has been done by the original architects. In exchange for this, there is time for the inexperienced programmer to learn how to use tools and to discuss and experience typical issues in the design, implementation, and maintenance of a large software system. The discussion of design principles becomes threaded throughout the course as the design decisions of the system authors are seen to influence the modification enterprise.

No attempt was made to evaluate the success of such an approach compared to that of a traditional course, nor are we proposing that our method be a substitute for a full-blown design and implementation experience, which we would urge as a successor to our course. Instead, we suggest that students without sufficient programming experience to *invent* designs can nevertheless absorb important principles of system design and engineering when they are cast as enhancements to existing systems. Subjectively, one may take as a measure of success the fact that the students were able to make a somewhat useful graphical interface to a mail reader in three weeks, despite the fact that none had had any experience with window systems before the project started. Also, those students looking for employment after the course quickly found good jobs. It is difficult, of course, to say whether this was directly influenced by the course or the fact that exposure to X windows programming made one especially hirable in the spring of 1988.

The only evaluation instrument applied to the course at all was a standardized student evaluation questionairre which is administered in all courses. The commentary on those forms made it clear that the students recognize that they knew more tools than when they started and that they were exposed to an important window system, but the fact that they also enhanced their craft is probably not entirely visible to them because its consequence is not very concrete.

Acknowledgements

The author wishes to thank C. H. Morris and the anonymous referees for suggesting substantial improvements in exposition and to one referee for correcting an important factual error about UNIX mail headers.

Reading List

Geoff Collyer and Henry Spencer, "News Need Not Be Slow," *Proc. 1988 Winter USENIX Conference*, pages 181-190.

Barry W. Boehm, "Improving Software Productivity," *IEEE Computer*, September, 1987, pages 43-57.

Richard J. Meyers and Jeff W. Parish, "The Macintosh Programmer's Workshop," *IEEE Software*, May, 1988, pages 59-66.

Richard M. Stallman, "EMACS, the Extensible Customizable, Self-Documenting Display Editor," in *Interactive Programming Environments*, David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, McGraw-Hill, New York, 1984.

References

Richard H. Austing, Bruce H. Barnes, Della T. Bonnette, Gerald L. Engel, and Gordon Stokes, editors, "Curriculum '78, Recommendations for the Undergraduate Program in Computer Science," *Comm. ACM*, 22(#3), March 1979, pages 147-166.

Catherine L. Bullard, Inez Caldwell, James Harrell, Cis Hinkle and A. Jefferson Offutt, "Anatomy of a Software Engineering Project," *Nineteenth SIGSCE Technical Symposium on Computer Science Education*, ACM SIGSCE Bulletin, 20(#1), 1988, pages 129-133.

Doris L. Carver, "Comparison of Techniques in Project Based Courses," *Sixteenth SIGSCE Technical Symposium on Computer Science Education*, ACM SIGSCE Bulletin, 17(#1), 1985, pages 9-12.

James S. Collofello, "Monitoring and Evaluating Individual Team Members in a Software Engineering Course," *Sixteenth SIGSCE Technical Symposium on Computer Science Education*, ACM SIGSCE Bulletin, 17(#1), 1985, pages 6-8.

David H. Crocker, *Standard for the Format of ARPA Internet Text Messages, RFC 822*, August 1982.

Lionel E. Deimel, Jr.,"The Uses of Program Reading," *ACM SIGSCE Bulletin*, 12(#2), 1985, pages 5-14.

John S. Quarterman and Josiah C. Hoskins, "Notable Computer Networks," *Comm. ACM*, 29(#10), October 1986, pages 932-971.

David S. Rosenthal, *A Simple Client Program, or How hard can it really be to write "Hello, World"?*, distributed with X11 release 2, 1988.

Marshall T. Rose, John L. Romine, *The Rand MH Message Handling System: User's Manual, UCI/UCB Version*, 1984, distributed with Berkeley UNIX documentation.

Robert W. Scheifler, James Gettys, and Ron Newman, *X Window System, C Library and Protocol Reference*, Digital Equipment Corp., 1988.

Kurt Shoens, revised by Craig Leres, "Mail Reference Manual, Version 5.2, April 1986," in *UNIX User's Supplementary Documents, 4.3 Berkeley Software Distribution*, University of California, Berkeley, 1986.

Richard M. Stallman, GDB Manual,*The GNU Source Level Debugger*, Free Software Foundation, Cambridge, MA, 1987.

Richard M. Stallman, *GNU Emacs Manual*, Free Software Foundation, Cambridge, MA. 1987.

Sun Microsystems, *Mail and Messages: Beginners Guide*, Part No: 800-1288-03, Revision A of 17 February 1986.

Dave Taylor, *ELM User Guide*, provided with ELM software. Copyright 1986, 1987 by Dave Taylor, Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304.

[*submitted Aug. 25, 1988; revised Nov. 21, 1988; accepted Dec. 16, 1988*]