CODING QUALITY AND TOOLS IN PROGRAMMING METHODS
A Report on Tool Novelty and Usefulness

Gary Perlman
School of Information Technology
Wang Institute of Graduate Studies

ABSTRACT

Twenty-two programming tools were introduced in coordinated exercises as part of a programming methods course. Twenty-two or twenty-four graduate students with work experience responded to a survey asking about previous and intended use of the tools. The survey showed that many tools were new and useful to the students. My conclusion is that it is worthwhile to Incorporate a module on coding quality and tools in software engineering programs.

1. Introduction

In Fall 1985 and Winter 1986, I taught a module on coding quality and tools (CQT) in the Programming Methods course. Programming Methods is a core course of the Wang Institute (WI) Master of Software Engineering (MSE) program, covering topics in design, coding, and testing. This module was added in response to faculty observations that some of our students produced low quality code in projects (e.g., bad style). The module had the following parts, with about one 90 minute lecture for each.

- Programming Environments (UNIX & C)
- Coding Style
- Code Comments and Documentation
- Static & Dynamic Analysis
- Pre-Processors & Efficiency
- Configuration Management

Some topics, previously single lectures in other courses, were drawn into the module because of heavy tool use and relation to program (rather than system) development.

I decided to build the module around software tools because I think that tools are an efficient way to improve programming practice and because student are motivated to learn about tools they might use elsewhere. Even if students did not have access to the UNIX/C tools covered in the CQT module, they would be in a better position to know if they would want to buy or build similar tools after having practical experience with them.

UNIX and C (Kernighan & Ritchie, 1979) were used for several reasons:

- It is a tool rich environment, the best we have at WI.
- It is used in many course projects at WI.
- It is used in industry, and gaining popularity.
- It was used in a concurrent Computer Systems Architecture class.
- I know it well.

2. The Tools

I obtained some new UNIX/C tools for the course:

cscope	code browser (UNIX toolchest at AT&T)
cstyle	coding style analysis (CACM via netnews)
scprof	statement count profiler (Catalytix Corp)

I wrote several new tools to aid the teaching of the module:

ccall	call graph cross referencer
cenv	programming environment template generator
elide	program elision
ff	text formatter and paginator
seec	program part extractor
shar	portable project archiver

The remaining tools include the classic UNIX programming tools:

cpp	C preprocessor
dbx	source code symbolic debugger
gprof	call graph timing profiler
indent	pretty printer
lint	program checker
make	program builder
rcs	revision control system
time	simple program timer

and some less used tools (like spell and tree).

More detailed descriptions of the tools are found in an appendix.

3. Exercises

To give the students practical experience with the tools, I devised a multi-faceted exercise throughout the module. I took a simple text formatting program, ff, and worked hard to clean it up as much as possible by making it portable, fast, and well styled. I used ff because its text formatting operations would be familiar to all the students: text filling, justifying, centering, indenting, and so on. While it was easy to understand what ff did, ff was a large enough program (about 650 lines of code with comments stripped out) to be non-trivial. If ff were much larger, there would have been too much overhead in learning about ff before doing the exercises.

I took the good ff, and degraded it in stages. Each version was a proper running version of the program. First, I made it slow by taking out some efficiency tricks and by doing some tasks stupidly. This was the slow version. Second, I made ff less portable to other systems by Inserting assumptions about how words are aligned and how function arguments are passed. This was the linty version, so named because it contained the sort of problems detected by the lint program. Third, I removed all the code comments, replaced some mnemonic variables names with cryptic ones, and randomly mangled the indentation. This was dubbed the ugly version.

The exercises were to work, aided by tools, back to the original ff. In the first exercise, students were given a paper listing of ugly ff, and had to comment on its style. They were also given the output of cstyle, run on ugly ff. Then they were allowed access to an online copy of linty ff. Exercise 2 had them use seec to extract subsets of the documentation from linty ff. The user manual and code comments for ff were embedded in specially tagged comments in the source file for ff. Exercise 3 required using lint and cscope on the linty ff source to find portability problems. Exercise 4 required using time, the gprof call graph timing profiler, and scprof statement count profiler, to find performance bottlenecks, and to improve them using rules from Bentley's book. The final report of Exercise 4 had to be archived (using shar) and mailed to me, showing that they had used rcs and make. Throughout this time, the students were encouraged to use other tools to aid their exercises. There were class demonstrations of most and discussions of all the tools.

4. The Survey

At the end of the CQT module, I distributed a survey to the students, asking them about their experiences with the tools used in the course. I wanted to know if I had gone over old material, and if I was introducing new tools, and did students plan to use them? I asked the following questions for each of the tools:

> How useful did you find the tool? (1-10 scale)
> Have you used the tool or one like it before? (Y/N)
> Do you plan to use the tool in the future (Y/N)

The module was taught to two classes:

Fall 1985	12 Full Time Students, 1 Part Time Student
Winter 1986	11 Part Time Students

Part time students tended to have more work experience than full time.

5. Results

Eleven of 13 Fall '85 students and all Winter '86 students responded to the survey. Students did not answer some questions about a tool, many writing that they were not familiar with the tool. All non-ratings for tools were dropped from analysis. The trends of the survey results were the same for both groups so the data were simply pooled. The major difference is that the part-time students, who tended to have more software development experience, also tended to be familiar with more tools.

A summary table of all the ratings is shown below. In the first column are the names of the tools, in decreasing order of rated utility. The tools are broken into two groups of 11 programs with subtotals being reported for the most useful and least useful programs. For each tool, there are total programmers rating whether they had used the tool (or one like it) before, and whether they planned to use it in the future. Marginal subtotals for each category are reported. Most of the rest of this report will discuss this table, so it is worth studying carefully.

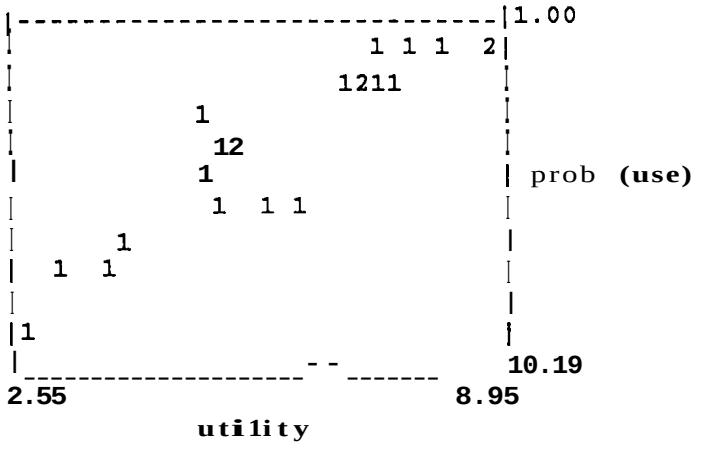
Used Before?		N			Y			Total		
Use Again?		N	Y	T	N	Y	T	N	Y	T
Tool	utility									
make	8.95	0	10	10	0	10	10	0	20	20
rcs	8.77	0	4	4	0	18	18	0	22	22
gprof	8.29	0	10	10	0	11	11	0	21	21
scprof	7.71	1	13	14	0	7	7	1	20	21
cscope	7.59	2	17	19	0	2	2	2	19	21
cpp	7.41	1	1	2	1	12	13	2	13	15
dbx	7.40	0	1	1	1	12	13	1	13	14
spell	7.13	0	2	2	2	18	20	2	20	22
time	7.10	2	4	6	0	15	15	2	19	21
lint	6.82	2	8	10	1	11	12	3	19	22
prof	6.25	3	4	7	4	5	9	7	9	16
Sub Tot	7.59	11	74	85	9	121	130	20	195	215
cenv	5.76	7	8	15	0	2	2	7	10	17
ff	5.53	5	8	13	1	5	6	6	13	19
xrf	5.38	1	0	1	3	11	14	4	11	15
ccall	5.12	2	3	5	2	7	9	4	10	14
shar	5.11	7	8	15	0	1	1	7	9	16
seec	4.95	7	11	18	0	3	3	7	14	21
tree	4.94	3	7	10	0	7	7	3	14	17
compile	4.00	6	1	7	0	4	4	6	5	11
elide	3.79	6	5	11	1	0	1	7	5	12
indent	3.07	7	2	9	2	3	5	9	5	14
cstyle	2.55	16	4	20	1	0	1	17	4	21
Sub Tot	4.56	67	57	124	10	43	53	77	100	177
Total	6.08	78	131	209	19	164	183	97	295	392

New Programs Introduced

Two people had previous experience with most tools, but most had varied experience. So just about everyone had no experience with several tools. Even though we accept only experienced programmers in the **MSE** program at WI--an average of **4-5** years--there were gaps in most student's tool experience. The results make me think that the time spent on the tools was well spent. Besides the experience with the tools and topics, they got experience with C and UNIX, and all this should improve the startup time in project courses. Of importance to me is the introduction of tools rated useful, and popular UNIX tools like make, and essential programming tools like profilers, were novel and well received.

Program Usefulness & Planned **Use**

As might be expected, there was a strong relation between rated usefulness of a tool and plans for future use. If we plot rated utility against the average rated probability of future use, we find a strong positive correlation (r = .93).



One result is that the students are not impressed that cosmetic style tools are useful to them (see the cstyle and indent programs). There were several popular new programs. cscope was used heavily in an Architecture course when the students tried to learn about an OS simulator. scprof and gprof seem to have fared well, and this is consistent with the popularity of the efficiency exercise and Bentley's book. People seemed to like ff, whose source code was used in all the coding exercises. My program contributions were met with mixed response, but I think it was worth writing them for class demonstrations, if only to give students experience that a tool does not seem useful (e.g., program elision). make was universally accepted. The totals show that of 209 novel tools, 131 (63%) are planned to be used in the future. For the top ranked (and best known) tools, this ratio is 74 of 85, or 87 percent.

6. References

Bentley, J. L. (1982) Writing Efficient Programs. Prentice Hall.

Berry, R. E. & Meekings, B. A. E. (1985) *A Style Analysis of C Programs.* Communications of the ACM, 28:1, 80-88.

Feldman, S. (1978) *Make - A Program for Maintaining Computer Programs.* Bell Laboratories.

Feuer, A. R. (1985) *The Safe C Dynamic Profiler.* Catalytix Corporation, Cambridge, MA.

Graham, S. L., Kessler, P. B., & McKusick, M. K. (1982) *gprof: A Call Graph Execution Profiler.* SIGPLAN Notices, 17:6, 120-126.

Johnson, S. C. (1978) *Lint: A C Program Checker.* Bell Laboratories.

Kernighan, B. W. & Ritchie, D. M. (1979) The C Programming Language. Prentice Hall.

Steffen, J. (1985) *Interactive Examination of a C Program with Cscope.* Proceedings of the 1985 Winter Usenix Conference, Dallas, TX. Usenix Association.

Tichy, W. F. (1982) *Design, Implementation, and Evaluation of a Revision Control System.* IEEE.

Appendix: Tool Descriptions

For each of the tools in the survey, I provide a source reference. If not in the list of references above, then there may be no published document other than the manual entry provided by the program author.

ccall (Perlrnan, **1985)**
ccall is a post-processor on the database created by cscope. ccall generates a full cross reference table or call graph for off-line printing.

cenv (Perlrnan, **1985)**
cenv is a C programming ENVironment template generator that helps automate and standardize the use of UNIX directories, makefiles, manual entries, module, and function headers.

compile **(McCready @** Tektronix)
compile finds specially tagged strings In files and executes them as UNIX commands. Usually, this string is a compilation command, but it is also used to run formatting commands on document files.

cpp (Kernighan **&** Ritchie, **1979)**
cpp is pre-processing pass of the C compiler. It allows file inclusion, conditional compilation, and macro substitution, with and without parameters.

cscope (Steffen, **1985)**
cscope is a screen-oriented, interactive C source code browsing tool that answers questions like, "Where is this function used?" and "What functions are called by this function?" It supports browsing and making global changes to many source files at once.

cstyle (Berry **&** Meekings, **1985)**
cstyle prints statistics on aspects of C programs thought to contribute to program readability. A weighted average score is computed based on Individual scores on aspects like use of indentation and blank lines, use of comments, use of goto's, etc.

dbx (Berkeley **UNIX** Manual)
dbx is an interactive symbolic source code debugger.

elide (Perlrnan, **1985)**
elide looks for (possibly nested) begin/end block markers and elides (removes) text nested more than some maximum depth. The begin/end markers might be program block delimiters, expression parentheses, comment markers, or string quotes.

ff (Perlrnan, **1985)**
ff is a simple text formatter that allows text filling with or without right justification at user specified width, line centering, pagination with user specifiable headers and footers, indentation, line numbering, variable line spacing, and tab-stop settings. As a programming tool, it is good for paginated program listings with line number and special tab stops.

gprof (Graham et al, **1982)**
gprof produces an execution profile of C, Pascal, or Fortran 77 programs. It uses call graph information to allocate time in called routines to the calling routines, thus providing more information than a "flat" profiler like prof.

indent (Berkeley UNIX Manual)
Indent is a source code beautifier that indents code lines, aligns comments, inserts spaces around operators where necessary, and breaks up cluttered declaration lists.

lint (Johnson, **1978)**
lint finds constructs likely to be bugs, non-portable or wasteful.

make (Feldman, **1978)**
make executes commands in a special file (a makefile) to update one or more targets. make updates a target only if it depends on files that have been updated since the target was last modified. These dependencies, and the commands to update targets, are encoded in the makefile.

prof (Berkeley UNIX Manual)
prof is a flat execution timing profiler, with functionality a proper subset of gprof.

rcs (Tlchy, **1982)**
rcs is a revision control system for saving and retrieving versions, file locking, and managing branches.

scprof **(Feuer, 1985)**
scprof Is a statement count profiler that reports for each executable statement line in a source file, how many times it was executed during one or more runs.

seec (Perlman, **1985)**
seec is a primitive C program parser that prints parts of C programs, such as identifiers, executable code, and comments. It can be restricted to print only those comments beginning with a special tag string, so typed comments can be made for function headers, declarations, algorithms, and so on.

shar (Perlman, **1985)**
shar is a portable file archiving system used for packing files, especially source files, to be mailed to many UNIX sites, some of which may not have shar. shar makes archives that the standard UNIX shell and supporting tools can unpack, thus insuring the portability of file and directory structure of programs sent through electronic mail to individual users or bulletin boards.

spell (Berkeley UNIX Manual)
spell is a minimal spelling checker that simply lists possible spelling errors on the user's terminal.

time (Berkeley UNIX Manual)
time reports the system and user CPU time used by a process.

tree (public domain)
tree prints a graphical display of a directory file structure.

xrf (Berkeley UNIX Manual)
xrf is a simple cross-reference generating tools that lists identifiers and lines in files where those identifiers are found.