The Software Design Laboratory

Jonathan M. Smith University of Pennsylvania

ABSTRACT: Software Design Laboratory is an undergraduate practicum in software design, which focuses on principles and practices of large-scale software design. Concepts and examples borrowed from elsewhere in Computer Science are applied to the construction of a significant project, namely a command interpreter resembling the Bourne shell. The course focus is on long-lived software systems of a size requiring group effort. We therefore address maintenance, testing, documentation, code readability, version control, and group dynamics.

1. Introduction

There is a transition in every Computer Science curriculum between introductory courses which are suitable for non-majors and more advanced courses. The former typically introduce one or more programming languages (often Pascal, but sometimes Lisp, Scheme, or some other language), touch upon basic data structures (e.g., trees and queues), and introduce fundamental algorithms (e.g., sorting and searching). The assignments are small: they either demonstrate language features or build toy applications of algorithms and data structures. They are, naturally, individual assignments. The latter courses, designed for Junior and Senior Majors, are typically electives, and offer in-depth treatment of some topic in Computer Science. These may range from Programming Languages, Operating Systems, and Artificial Intelligence to Analysis of Algorithms, and tend to reflect faculty interests more than introductory courses. The more practically-oriented courses often use projects or case studies to reinforce concepts discussed in class. The instructors may stimulate the advanced undergraduate to participate in research efforts by means of a project course or a directed independent study. Many of these projects use the C programming language, or a derivative such as C++, and expect a working knowledge of the UNIX operating system. The work in such courses may be done in groups, or it may be done in collaboration with active researchers who have significant software efforts. At a high level, this describes the situation at Penn and many other schools[†].

We have developed a course in *software design* which we believe fits well at the point of the transition. We call this course a "Laboratory" for its training in the *application* of principles. In this, it is like laboratories offered by other disciplines such as Physics and

† This course started at Columbia University and has continued to evolve at the University of Pennsylvania.

Chemistry. Unlike traditional laboratories, the focus is less on the experimental method than on learning from a single extended experiment. The learning is directed towards construction of significant long-lived systems, as opposed to construction of throwaway examples. A number of observations helped shape the course:

1. Significant software engineering tasks have a long lifetime, characterized by a design phase, an implementation phase, and a long "maintenance phase." In real systems, the "maintenance phase" accounts for most of the money spent, and thus there is typically significant effort spent in the design phase to ease maintenance. One difficulty with long-lived systems is that environments change and new features are required. Thus, one must design for maintenance, coupled with the notion of software re-use. A course should structure assignments in such a way that previous work must be reused, as in an implementation done in phases. At each phase, the previous code is used, or the instructor's code is used (necessitating reading and understanding a system which is more complex as time goes on), as a platform.
2. Testing design strategies which enable and ease testing must be introduced. In many cases, design activities are essentially independent of an implementation, but all implementation phases demand testing. The choice of test cases, and the choice of testers, is crucial to effective testing.
3. Documentation is essential, because many involved in the design and engineering of significant systems do not want to read the precise statement of problem solution embodied in the code in order to obtain adequate understanding for their role. They want to understand precisely only the interfaces required for performance of their own tasks. The rest may be useful for the big picture. This documentation can take many forms:

    - embedded commentary
    - associated files in a text-processing language such as *troff* or $\TeX$.
    - pointers to relevant literature embedded as comments in the program text
    - Roadmaps or meta-documents describing relationships between modules

All of these things aid human understanding, because big systems need more than a few people on the same wavelength.

4. Coding standards, even loose ones, help program readability. It is important to read code, and code must be written in such a way as to be read; style sheets help this.

The course has successfully accomplished these goals for a number of semesters. The student leaves the course with a thorough understanding of a tool-rich programming environment that many professional programmers consider an excellent one. More important, they will have worked on a project of significant scope, built a significant software artifact, and will understand the group nature of systems building. Formal methods are addressed as a methodology, not a known solution.

Software Design Laboratory ("Software Lab") is an undergraduate course, and thus differs significantly from graduate-level software engineering training, e.g, Wang Institute's now-defunct [Ardis 1987, McKeeman 1987] Master of Software Engineering (MSE) program. For such courses a higher level of prerequisites and background could be expected, and sufficient attention paid to all aspects of the software lifecycle. Such graduate courses presumably have the advantage of prior student exposure to Computer Science, and thus can direct more energy towards software engineering, and less towards the "glue" connecting software design to other areas of Computer Science. Some of these other courses, in particular the "Software Hut" [Horning & Wortman 1977, Wortman 1987], have addressed group structure and interaction issues in a different fashion than Software Lab, but for the thrust of our course these differences do not seem appropriate. An interesting observation is made in the 1987 article [Wortman 1987] on the Toronto course, where Wortman states: "We now feel that the emphasis on buying and selling software in the original software hut project gave the whole project the wrong orientation. The course we teach is about the design and implementation of software, not about software marketing." Kant's [Kant 1981] course, with students ranging from freshmen to graduate students covers different portions of the life cycle than Software Lab. Her article provides a course outline, with interjected textual comments. The feedback was similar; namely, the course required too much work for the number of credits. Her group size was 5, versus our 3.

Software Lab is consistent with the survey results gathered by Leventhal and Mynatt [Leventhal & Mynatt 1987] in that it is offered to Junior and Senior-level students, focuses on "Later-Life-Cycle" issues, is project-oriented, the grade is heavily based on success with the project, and the substantial project is intended for actual use. We differ in that the requirements for written reports are lessened (this stems partly from the project, an existing well-documented piece of software) and no oral reports or examinations are required.

Bentley and Dallen's [Bentley & Dallen 1987] setting is similar, although their course offering appears to be slightly later in the West Point curriculum than Software Lab is in ours. We note their approach of using many smaller exercises to teach software engineering principles. This contrasts with Software Lab approach of using a single large project, partitioned into development stages.

Morris's [Morris 1988] course is very similar to Software Lab; he recognizes many of the same needs, and took similar approaches. The major difference we see was the choice of project, a mailer, versus Software Lab's command interpreter (discussed in the second section). Since the command interpreter is a programming language, and its functionality is tightly integrated with the features of UNIX, our exercise effectively bundled up learning experiences from several domains. As we argued earlier, this effectively integrates a software design practicum with other portions of our Computer Science curriculum. Thus, it both builds upon and reinforces that curriculum.

The remainder of the paper is organized beginning with a rather detailed presentation of course material in the second section. The course is summarized in Table I at the end of the section. The third section discusses the course management issues and relates Software Lab to laboratory exercises in classical scientific disciplines. The fourth section concludes the paper and relates the course's accomplishments to its educational goals.

Description	Role	Readings
1. Associative Memory values in in-core DB, add persistence	Familiarize with UNIX, C, simple parsing & I/O, reading code, comments	[Bourne 1978] [Ritchie & Thompson 1978] Style Sheet
2. Change shell variables passed to subcommands	Pathnames, processes, design from specification	[Thompson 1978]
3. Describe approach to problem in a report	Force understanding of problem	[Kerninghan & start group work]
4. Single-process execution with redirection syntax	learn *lex*, *yacc*, UNIX I/O calls, *fork()*/*exec()*, SCCS	
5. Argument pattern matching	Directory structure, regular expressions, reuse, testing	[Brooks 1975]
6. Writer Out → Reader In named string variables	Concurrency, IPC, Macro substitution, string manipulation	
7. " - execute string "" - escape white space " -escape everything	Escapes, regression testing recursive processes, parser state variables	[Weinberg 1974]
8. Abstract from mistakes & successes in document	Learning principles from examples	

Table I: Summary of Course Phases

2. *Course Details*

The course presentation is designed so that covered material would not become obsolete upon completion of the course; there is development of both a project and a general purpose *toolbox,* of both code and techniques.
The following books comprise the course reading list:

- *The UNIX Programming Environment* [Kernighan & Pike 1984], chosen because it illustrates use of the UNIX tools and libraries on a realistic example, namely a small programming language.

- *The Psychology of Computer Programming* [Weinberg 1974], chosen because it focuses on the fact that programming (software design) is a human activity, and that as the size and complexity of the system increases, the nature of the proper support tools changes from programmer support tools to group support tools. Also stresses reading programs, and "egoless programming" (groupthink). Batch programming discussion is unfortunately a bit dated.
- *The Mythical Man-Month* [Brooks 1975], was chosen for its readable and insightful discussion of the OS/360 software development and lessons learned. While many points echo Weinberg, chief programmer teams are quite different than egoless programming.

In addition, the following books and articles are background reading: "The UNIX Operating System" [Ritchie & Thompson 1978], "The UNIX Shell" [Bourne 1978], "UNIX Implementation" [Thompson 1978], and *"The C Programming Language"* [Kernighan & Ritchie 1978] is also suggested for students unfamiliar with C and UNIX:

In the next eight subsections, we present the assignments that are given and their intended role. All assignments involving programming are specified as a UNIX manual page, a clear and concise form of specification that the student is to be familiar with. An example manual page for a programming assignment is included as Appendix I.

2.1 Associative Memory

The first order of business is proficiency in writing, and especially in *reading* the language used in the course, C. The students are advised to consult Kernighan and Ritchie [Kernighan & Ritchie 1978] and are given a "Style Sheet for C" which suggests a stylistic convention for writing C source and building well-documented multi-module programs.

A program implementing an "associative memory" is distributed to the class, in source form. The program prompts the user for an input; the input is a new-line terminated string of characters. If the input contains a '=' character, the characters to the left of the '=' are treated as a *name* and the characters to the right are treated as a *value,* which is associated with that name. If there is no '=', and the input contains a '$' character, the characters to the right of the '$' are treated as a *name;* the associated *value* is retrieved and printed if there is one. If neither '=' or '$' are present, the program merely prompts for another input. It accepts input lines until an end of file condition is raised. The <*name, value*> pairs are stored as singly linked lists of structured records.

Thus, reading the well-commented source code introduces the students to strings, records, terminal I/O, simple parsing, subroutines, dynamic memory allocation, and pointers (always a source of trouble to the student). The lecture material emphasizes the necessity of reading source code. Using the conventions of the style sheet helps to *write* readable source code.

The assignment is to modify the program so that it preserves *name, value* pairs across invocations, i.e., it maintains them on disk storage. This introduces the student to operations on named disk files, and forces an understanding of the list maintenance code.

2.2 Env Command

Other than the file operations required to manipulate the <*name, value*> pairs across invocations, the student has encountered little of UNIX. The second assignment is the *env(1)* command, which is available with System V UNIX, but not with most versions of 4.[X] BSD, which is used for teaching. The *environment* is a set of <*name, value*> pairs that are made available to subprocesses; it is a subset of the <*name, value*> pairs accessible to the shell user. It provides a method for users to pass information to subprocesses without explicitly specifying options on a command line, e.g., the terminal is specified with TERM=hp2621; all screenoriented programs examine this value to determine appropriate terminal control sequences. The assigned *env* command has the invocation syntax:

env [-] [name=value]* [command [argument]*] where containing brackets indicate that the contents are optional, and "*" is the usual Kleene star, indicating zero or more repetitions. The command argument specifies a UNIX command to execute. With no command argument, the program prints the strings contained in the current environment, otherwise the command is executed with the specified string settings in its environment. The name=value arguments specify new settings, and the "-", if present, specifies that the current environment is to be ignored.

The program added the following to the students education:

1. Understanding of the UNIX command line argument handling discipline. Thus, simple parsing is covered.
2. Process management, since the mechanism for setting the environment values uses the *exec()* system call.
   3.Further understanding of the file system, since command lookup required search through several directories, specified through the PATH environment variable.

In addition, the student is able to make use of whatever string management utility routines they had developed for the first assignment.

2.3 Design Document

The first two assignments are to be done individually; they are exercises to ensure sufficient exposure for contributions in a group setting. The students are assigned readings describing the command interpreter [Ritchie & Thompson 1978, Bourne 1978] whose subset would be implemented. Groups are formed; students are allowed to form 3-4 person groups with their acquaintances; groups of the remaining individuals are formed at random; the ideal size is 3.

Given their readings, the students are requested to submit a design document describing their approach to designing the program described in the literature. This is done both to ensure that they had read the literature and to create some group cohesion; there is no intention to hold them to the design. They are expected to detail data structures, algorithms, and user interface features. At this point, they are introduced to several powerful UNIX tools for program construction, *make,* a dependency-specifying tool for recompilation; *lex,* a lexical analyzer generator; and *yacc,* a parser generator. While they are given appropriate readings, a more effective tool is to give them an example. The example is the first assignment redone using the tools; experience with the assignment helped the students to see the value of these tools.

2.4 Command Execution and I/O Redirection

The first iteration of command interpreter development required that the student provide an interactive facility for executing commands with arguments and specified I/O redirections. These redirections allow commands operating on the standard output and input files to have the file values specified on the command line. The syntax provides mechanisms for reading, writing, and appending to named disk files, as well as the ability to operate on previously opened files specified by a small "file number." There is additional syntax for interactive entry of files immediately previous to command execution.

The assignment allowed the students to use the mechanisms developed in the *env* assignment to create an interactive command interpreter. The new learning consisted mainly of the use of the tools, which for a first-time user is non-trivial. Their understanding of file manipulation technique is expanded greatly.

2.5 Metacharacters for Filename Pattern-Matching

The second version of the command interpreter added metacharacters to the command line syntax. Metacharacters, e.g. the wild card character "*", are used to pattern match filenames so that lists of arguments can be specified in a compact fashion. For example, "`pr *.[ch]`" will print the C source files and headers in the current directory. These patterns can be arbitrarily complicated; see Bourne [Bourne 1978] for details. The design of these additions involved several components, of which the most important are a pattern matcher and an interface to the UNIX directory structure, so that multi-directory patterns such as "`/u*/faculty/j??/t[12]*`" could be properly evaluated.

Class time is spent on regular expressions and metacharacters, e.g., the Kleene '*'. Once the regular expression notion is understood, the construction of a pattern matcher became an exercise in coding. The students are advised to first implement a single directory pattern expansion routine, which could then be recursively applied to the multiple directory case. Thus, the students are exposed to:

1. Regular expressions (which they had first encountered with *lex*), and more significantly, their implementation.

2. Pattern matching algorithms.
3. Hierarchical file systems.

The effect of this exposure is very positive, in that the student sees the advantage of such compact notations as regular expressions, and the simplicity and power of the hierarchical file system in a *practical* setting.

An important feature of the approach is the integration of new features into an existing software framework. Thus, good design decisions and engineering practice, e.g. documentation, pay off in later assignments. Poor decisions make integration more difficult, and may force substantial redesign. Thus the students are exposed to the issues of software maintenance in a most practical fashion.

2.6 Multiprocess Computations and Symbol Manipulation

In the third iteration, there are two additions to the command interpreter. These are the addition of syntax and functionality for connecting processes via pipes, and inclusion of facilities for setting and retrieving named string-valued variables.

This assignment posed particular conceptual problems for the students; we attribute it to their first encounter with *concurrency,* virtual or otherwise. Use of the *fork()* primitive in previous exercises helped, but less than it might have since they are given a canonical code segment containing the common *fork()/exec()* sequence. The inclusion of facilities for variables drew on their earlier experiences with the "associative memory"; many groups re-used the code.

2.7 New Parsing and Execution for "Quotes"

The fourth and final additions to the command interpreter are the three types of quotation marks employed by the UNIX Shell, single quotes ('), back-quotes (`), and double quotes (") [Bourne 1978]. This addition is chosen for the following two (major) reasons:

1. It forced a careful redesign of the lexical analysis routines and their interface to the parser and interpreter. Other than to add "|", the symbol for separating pipeline components, there had

been no changes necessary to the lexical analyzer since the initial assignment.

2. The implementation of the back-quote, which specifies a string-valued result to be obtained by executing the contained commands, forced the students to *glue* things together carefully. In particular, the easiest way of implementing this feature is with a copy of the command interpreter invoked through a pipeline.

Attention is given to issues such as the order of evaluation applied to the various features, and the demands this made on the implementation strategy, for example the command string "`a=*; echo $a`". Progress through the programming assignments towards the complete project is illustrated in Figure 1.

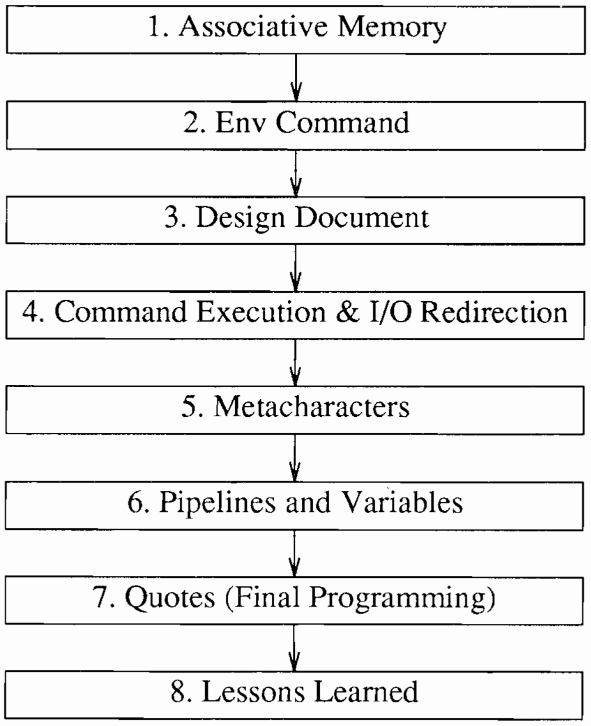
1. Associative Memory
2. Env Command
3. Design Document
4. Command Execution & I/O Redirection
5. Metacharacters
6. Pipelines and Variables
7. Quotes (Final Programming)
8. Lessons Learned

Figure 1: Steps towards final project

2.8 Lessons Learned

Mistakes (and triumphs), in retrospect, are among the most valuable learning experiences. Accordingly, the students submit a "Lessons Learned" document, summarizing their positive and negative experiences with tools and methodologies. In order that they understand what such a document is to contain, a realistic example is given based on the instructor's problems in constructing the command interpreter. As always, there is a wide separation between the best and worst of these documents; the best are remarkably frank and insightful, while the worst are obvious or mere restatements of the distributed example.

What is most exciting is that many students discover and formulate principles of good design and debugging methodologies for themselves, with examples they have taken to heart because they had built them.

Table I gives a summary of the course phases shown in Figure 1. Phases 1-2 are individual effort, and Phases 3-8 are group effort. The final project is complete by Phase 7, and Phases 3 and 8 are external documentation steps. Project construction is from Phases 3 through 7. Phases 1,2,3,5 and 8 are allowed one week for completion; Phases 4, 6 and 7 are allowed two weeks. In practice, the course schedule may slide a bit during the semester, but adjustments are easily made. Figure 2 illustrates the major reuses of code by relating reuse to the implementation phases of Table I. The relationship is illustrated by enclosure; if Box N encloses Box M, Phase M's code was used in Phase N.

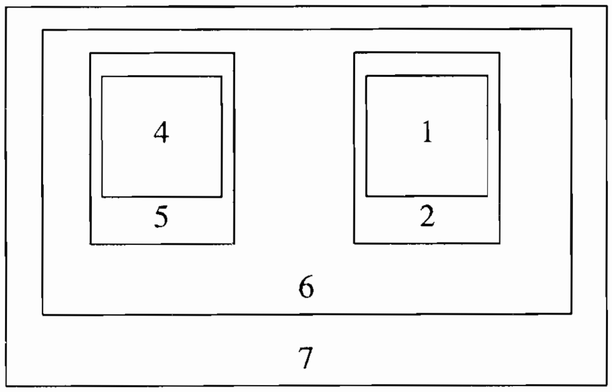

Figure 2: Re-use between implementation phases

3. Discussion of the Course

There are several important components we see in a laboratory setting, namely (1) experiments; (2) replication of experiments; (3) observation and deduction; and (4) "classical" laboratory techniques, such as maintaining laboratory notebooks or logs.

The course met for two sessions per week. The first session is interactive, and the second is in a lecture format. This ordering can take advantage of an intervening weekend to stimulate questions; students experimented with the material presented in the second day's lecture. The lecture material emphasized *testing* and *observation* of the results; a terminal in the classroom is used often. Office hours and help sessions were held in areas with terminals. Student experimentation is of two types. First, given that the students were implementing a shell-subset command interpreter, they could resolve questions about the intended functionality of their software in a simple fashion. In their "reverse-engineering", they could experiment with the standard shell to test the behavior of redirection and quotation marks. The *deductions* drawn from these experiments were incorporated into the design of the student projects. Students were enthusiastic about experimenting; their experiments detected mistakes in preceding lectures! Second, the students experimented with new concepts by writing small programs. For example, a trivial multiprocess pipeline was implemented to understand synchronization and data movement. This method of experimentation is the basis for *prototyping*.

Experiments were replicated by the students for several reasons. First, in debugging, a failure must be repeatable to be isolated and diagnosed. Second, many groups performed experiments suggested in class to increase their grasp of the material. Discussion between students led to many unexplained phenomena arising as questions in the next interactive class session.

Observation is dealt with in three ways. First, several lectures and interactions dealt with the experimental methods necessary for reverse-engineering a large program. Second, the process of debugging software was discussed. Some general principles of observation, fault-detection, and fault-refinement were given. Third, a detailed lecture on performance measurement and analysis was given. This took a paper from the scientific literature, explained the results and procedures, and then examined the conclusions. The lecture emphasized measurement, presentation, and the validity of conclusions.

"Classical" lab techniques were not always applied, as the setting is not the physical sciences. One technique deserving attention is record maintenance. Suggested documentation included source-code comments describing methods, and measurements justifying design decisions, e.g., use of a certain method. Thus, the comments existed as a record of the design decisions and motivations. The "Lessons Learned" document served as a summary record of the student's observations; some of these were surprisingly detailed.

We used electronic communication extensively; this allowed the student to obtain answers across the week, rather than a few preset times. An on-line bulletin board mechanism allowed posting of sources, interesting questions, interesting answers, and details of the assignments. This saved class time for more appropriate interactions. The choice of an existing software system had a number of positive effects, including:

1. The command interpreter they were constructing is documented completely [Bourne 1978]. Such command interpreters are (1) interactive, (2) programming languages, and (3) interfaces to an underlying operating system, which provides a virtual machine. In addition, the shell is an exemplary piece of software design.
2. The *full* interpreter they were working towards is the student's interface to the system. Thus, they become familiar with its functioning through *use* as well as instruction. Questions about obscure functional details could be answered by typing in one or more well-chosen examples. Experimentation was a very worthwhile tool, as it should be in a laboratory course. Several groups of students corrected the instructor on interpreter details based on their independent experiments (sometimes success can be embarrassing!).

The instructor completed all assignments, and generally made the results available on-line. This (1) gave feedback on the complexity of the assignments; and (2) gave enough insight and mastery of detail to aid the student in all phases of the design process.

Grading of all programming assignments previous to the project completion relied on an even split between code quality and execution testing. The execution testing was done based on the manual page used to specify the assignment, and the evaluation of code quality had both an objective portion, consisting of adherence to a style sheet, and a subjective component, based on the grader's judgment. The effect of the subjectivity was reduced by dividing the assignments between the instructor and the teaching assistants, with the division occurring randomly on any given assignment. The final project was graded wholly by success or failure on a set of 30 tests designed to exercise the features specified in the manual pages. Thus, the quality of the student's results were reviewed. Subjective performance measures, such as effort expended, or document formatting skill, were not involved. This is as it should be. One difficulty which seems to always occur in group work is unequal contributions. This was resolved by assigning all group members the same grade unless there was a complaint. If there was a complaint, the entire group was required to be present to discuss reassignment of credit. Those not present at the discussion were assumed to be in agreement with whatever conclusion was reached. This resolved all complaints in a satisfactory manner.

4. Conclusions

Aside from introducing the students to C and the UNIX programming environment, the course structure has several strong points:

- The student develops a non-trivial *toolkit,* consisting of both techniques and developed skills with software tools.
- The focus on one significant project brings out the point of software engineering, which is only apparent with scale and re-use (much like civil engineering versus home carpentry).
- The process of building the project is used both to get across the introductory material (in the individual assignments) and to bring in classical software engineering issues, such as documentation, tool usage, maintenance, reusability, *et cetera.* In particular, forcing integration of new features with previous work demands that attention be paid to *design.* Of course, building on previous work shows the value of *re-use,* as illustrated in Figure 2.

- The course is a lab course, and thus is exceedingly *practical* in orientation; discussion of issues such as the communication problems and solutions of Brooks [Brooks 1975] are postponed until the student has encountered them, and can appreciate the solutions.

Discussions with faculty colleagues reinforce the belief that the toolkit approach has value in this setting; a discussion of lexical analysis and parsing certainly makes more sense when the student has already encountered these topics in practice; with some practical exposure, history, current approaches and theory not only become more accessible but more relevant.

The results have been encouraging in many ways, but work remains to be done. The graduates of the course have, on the one hand, been well-prepared for project courses and work on faculty research projects, as well as for jobs. On the other hand, there is a real risk that a practical course can acquire a "trade school" orientation, and the instructor must ensure that material of lasting value is taught. It is too easy to focus on technological details, and often hard to discern true principles from folklore. It takes time, and we are still learning.

5. Notes and Acknowledgements

Rodney Farrow, Steven Feiner, John Ioannidis, Gerald Leitner, Gerald Q. Maguire, Jr., and Peter Sweeney helped develop and refine Software Lab with the generosity of their ideas and time. The students and teaching assistants of Software Lab fed back many learning experiences into the current course.

AT&T donated many copies of the "UNIX" Issue (July-August, 1978) of the Bell System Technical Journal used in several editions of Software Lab, and equipment donations from AT&T, Hewlett-Packard, and IBM have supported the computing needs.

A number of anonymous referees provided useful suggestions.

The complete materials for the course (which changes slightly every year) are available via anonymous FTP from `dsl.cis.upenn.edu`, in the file `~ftp/pub/sdl.tar.Z`.

References

M. A. Ardis, "The Evolution of Wang Institute's Master of Software Engineering Program," *IEEE Transactions on Software Engineering,* Vol. SE-13(11), pp. 1149-1155, Special Issue on Software Engineering Education (November 1987).

J. L. Bentley and J. A. Dallen, "Exercises in Software Design," *IEEE Transactions on Software Engineering,* Vol. SE-13(11), pp. 1164-1169, Special Issue on Software Engineering Education (November 1987).

S. R. Bourne, "The UNIX Shell," *The Bell System Technical Journal* **57**(6, Part 2), pp. 1971-1990 (July-August 1978).

F. P. Brooks, Jr., *The Mythical Man-Month,* Addison-Wesley, Reading, Mass. (1975).

J. J. Horning and D. B. Wortman, "Software Hut: A computer program engineering project in the form of a game," *IEEE Transactions on Software Engineering* **SE-3,** pp. 325–330 (July 1977).

E. Kant, "A Semester Course in Software Engineering," *ACM SIGSOFT Software Engineering Notes* **6**(4), pp. 52–76 (August, 1981).

B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall (1978).

B. W. Kernighan and R. Pike, *The UNIX Programming Environment,* Prentice-Hall (1984).

L. M. Leventhal and B. T. Mynatt, "Components of Typical Undergraduate Software Engineering Courses: Results from a Survey," *IEEE Transactions on Software Engineering* **SE-13**(11), pp. 1193-1198, Special Issue on Software Engineering Education (November 1987).

W. M. McKeeman, "Experience with a Software Engineering Project Course," *IEEE Transactions on Software Engineering* **SE-13**(11), pp. 1182–1192, Special Issue on Software Engineering Education (November 1987).

Robert A. Morris, "An Unorthodox Approach to Undergraduate Software Engineering Instruction," *Computing Systems* **1**(4), pp. 405–419 (1988).

D. M. Ritchie and K. L. Thompson, "The UNIX Time-Sharing System," *Bell System Technical Journal* **57**(6), pp. 1905–1930 (July-August 1978).

K. L. Thompson, "UNIX Implementation," *The Bell System Technical Journal* **57**(6, Part 2), pp. 1931-1946 (July-August 1978).

Gerald Weinberg, *The Psychology of Computer Programming,* Van Nostrand (1974).

D. B. Wortman, "Software Projects in an Academic Environment," *IEEE Transactions on Software Engineering* **SE-13**(11), pp. 1176–1181, Special Issue on Software Engineering Education (November 1987).

[submitted July 6, 1991; revised Aug. 24, 1991; accepted Sept. 9, 1991]

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the *Computing Systems* copyright notice and its date appear, and notice is given that copying is by permission of the Regents of the University of California. To copy otherwise, or to republish, requires a fee and/or specific permission. See inside front cover for details.

Appendix I
EXPAND(3)

NAME

expand()-file name generation routine

SYNOPSIS

char **expand(word)
char*word;

DESCRIPTION

*expand*is used to provide the file name generation facilities described in *sh(1)*. The argument *word* is a null-terminated string of characters. If any of the three characters *, ?, or [is contained in *word, word* is regarded as a *pattern. expand()* returns a list of pointers to alphabetically sorted file names that match the pattern; the list is terminated by a NULL character pointer. If no file name is found which matches the pattern, *expand()* returns the list consisting of a pointer to *word* and the NULL pointer. The character . at the start of a file name or immediately following a /, as well as the character / itself, must be matched explicitly. * Matches any string, including the null string. ? Matches any single character. [...] Matches any one of the enclosed characters. A pair of characters separated by - matches any character lexically between the pair, inclusive.

EXAMPLES

```
expand("*.[ch]");
expand ("/usr/faculty/jms/*.d/[a-z]*.?");
```

USAGE

expand() should be incorporated into your previous assignment, *io(1)*, so that input lines containing patterns should be executed correctly, e.g.

```
    $ echo * >file1 >file2
```

should create an empty file1 and an alphabetically sorted list of file names from the current directory should appear in file2.