# A Practical Course in Software Design

*Jonathan M. Smith*

Columbia University Computer Science Department, New York, NY 10027

Technical Report CUCS-357-88

*ABSTRACT*

In practical disciplines, ''Those who can, do. Those who can't, teach.'' and you ''Learn by doing.''. Our presentation of an undergraduate semester course in Software Design, ''Software Design Laboratory'', has the spirit of the second adage and attempts to refute the first.

In our description of the course, we focus on the relationship between the different programming assignments, and the role of these assignments in developing the student's capabilities, rather than on management, group structure, or formal techniques. We argue that a laboratory course is as essential to Computer Science as it is to Physics or Chemistry.

Keywords: Software Engineering, Education, Software Design

# A Practical Course in Software Design

*Jonathan M. Smith*

Columbia University Computer Science Department, New York, NY 10027

Technical Report CUCS-357-88

## 1. Introduction

One of the great difficulties in Computer Science (CS) education is the integration of material of a practical nature with the more abstract material. Here, we report our approach, that of a "Software Design Laboratory" (abbreviated SDL), where much like the Laboratory courses offered by other disciplines such as Physics and Chemistry, training in the *application* of principles is emphasized over the presentation and dissemination of the principles. In the terminology used by Leventhal and Mynatt [11], we focus on the "Later-Life-Cycle", as opposed to the "Early-Life-Cycle" or "Theoretical-Issues" approaches; these are functions of another course offering. We compare our approach with related work after a discussion. The remainder of the introduction describes the CS curriculum at Columbia, the role of the SDL course in this curriculum, and a short history of recent versions of the course.

### 1.1. Curriculum

The required courses [15] for a CS major in the School of Engineering and Applied Science at Columbia are as follows:

Introduction to Programming
Discrete Mathematics, I: Introduction to Combinatorics and Graph Theory
Discrete Mathematics, II: Introduction to Discrete Structures
Applied Mathematics, I
Introduction to Probability
Data Structures
Fundamental Algorithms
Software Design Laboratory
Digital Logic
Computer Organization, I
Computability and Models of Computation
Programming Languages and Translators
Artificial Intelligence
Scientific Computation, I
Computer Organization, II
Operating Systems
Computer Networks and Data Bases

In addition, students are required to take several technical electives; a large number of advanced courses, project courses, and seminars are available to fulfill this requirement.

### 1.2. SDL Role

The courses are listed in (approximately) the order they are intended to be taken, although course and student scheduling constraints may perturb the ordering. The material encountered previous to SDL is mostly abstract, and Fundamental Algorithms can be taken concurrently with SDL. The students are exposed to Pascal in the introductory course, and that language is used in Data Structures and Fundamental Algorithms when programming assignments are given. The programming assignments in these courses are typically small, and designed to expose features of isolated data structures, algorithms, or programming languages. In the more advanced software courses, such as Programming Languages and Translators, and Operating Systems, knowledge of the UNIX® operating system and the C programming language [9] are presumed. Almost all of the project courses require experience with UNIX, and LISP or C.

Given this organization, it's clear that at some point in the curriculum, the student must be prepared for the move from the Pascal-instructed isolated program mode to one based on C and UNIX where

⇒ A version of this paper has been submitted for publication. Please contact the author for references to this work.

This work was supported in part by an equipment grant from the Hewlett-Packard Corporation and NSF grant CDR-84-21402.

® UNIX is a registered trademark of AT&T Bell Laboratories

significant programs are written. The role of the SDL course in our curriculum is then:

- To expose the student to their second programming language, C.
- To familiarize the student with the UNIX operating system.
- To teach the students how the construction of large software projects is accomplished.
- To anchor CS concepts with well-chosen and relevant examples.

The course has successfully accomplished these goals for many semesters.

### 1.3. History

SDL has been in the curriculum for many years, and is still evolving. It was originally a more advanced course offering, but the more abstract aspects of software engineering (the "Early-Life-Cycle" and "Theoretical-Issues" material) are now separated into an elective course in Software Engineering. The author served as a Teaching Assistant in three previous versions of the course before instructing the course; the historical data is derived from that experience.

1. (1982) The course was taught using Brooks' "Mythical Man-Month" [5] as the primary text. The in-class lectures were derived from this book and Kernighan and Plauger's "Software Tools"; there were three assignments, rather loosely connected with the texts and in-class presentation. These were a macro processor (in the style of M4 [8] ), a simple command interpreter based on the UNIX shell [4], and an implementation of the UNIX *pipe()* system call which allowed the buffer size to be set dynamically by the caller.

   The presentation of the solutions was typically as a *design*, rather than an implementation, and documentation was an integral part of each assignment. The implementations were done under Amdahl's UTS™ operating on an IBM 4381™ mainframe.

2. (1984) The same textbooks, a new instructor, and new hardware were combined in a somewhat different presentation of the course. The available hardware running UNIX was a set of approximately twenty AT&T 3B2/310™ supermicrocomputers, with attached AT&T DMD5620™ bit-mapped display terminals. The instructor's interests were in operating systems and graphics; a single course project was assigned, emulation of the MacDraw™ or MacPaint™ programs for the Apple MacIntosh™ computer. The material covered in class was somewhat disjoint from the project orientation of the course; the Teaching Assistants were occasionally asked to give short lectures on the interface to the graphics terminal or operating system features. There was a strong emphasis put on documentation, and the programs were graded based on the quality of the documentation as well as the performance at a public demonstration.

3. (1985) This was essentially 1984 *redux* with a slightly different graphics project.

## 2. Description of the Present Course

The course presentation was designed so that covered material would not become obsolete; the student would be constantly working towards both the development of a project and a general purpose *toolbox*, of both code and techniques, which would serve them well in both this course, and later courses. In the next subsections, we present the assignments that were given and their intended role in the student's experience. All assignments involving programming were specified as a UNIX manual page, a clear and concise form of specification that the student was to be familiar with.

### 2.1. Associative Memory

Since the students were not expected to be familiar with C, but had experience with another Algol-derived language, Pascal, the first order of business was proficiency in C. The students were advised to consult Kernighan and Ritchie [9] and were given a "Style Sheet for C" which suggested a stylistic convention for writing C source and building well-documented multi-module programs. The textbook used for the course was Kernighan and Pike [10].

A program implementing an "associative memory" was 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

UTS is a trademark of Amdahl.
4381 is a trademark of IBM.

3B2 and DMD5620 are trademarks of AT&T.
MacIntosh, MacDraw, and MacPaint are trademarks of Apple Computer.

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 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

Other than the file operations required to manipulate the *<name, value>* pairs across invocations, the student has seen relatively little of UNIX. The second assignment is to implement the *env(1)* command, which is available with System V UNIX, but not with Ultrix™, which we use for teaching at Columbia. 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., my terminal is specified with `TERM=hp2621`; all screen-oriented programs examine this value in order 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. 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 contributed the following tools to the students kit:

1. Understanding of the UNIX command line argument handling discipline. Thus, simple parsing is covered.
2. Process management[1], since the actual mechanism for setting the environment values is with 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 was 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 were to be done individually; they were exercises to ensure that all students had sufficient exposure to contribute in a group setting. The students had been assigned readings which described the command interpreter which they were going to be constructing a subset of; these were Bourne [4] and Ritchie and Thompson [13]. Groups were formed; students who knew each other were allowed to form 3-4 person groups; groups of the remaining individuals were formed at random; the ideal size was 3.

Given their readings, the students were requested to submit a design document describing their approach to designing the program described in the literature. This was done both to ensure that they had read the literature and to create some group cohesion; there was no intention to hold them to the design. They were expected to detail data structures, algorithms, and user interface features. At this point, they were 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 were given appropriate readings, a more effective tool was to give them an example. The example was the first assignment redone using the tools; experience with the assignment helped the students to see the value of these tools.

Ultrix is a trademark of Digital Equipment Corporation

[1] As can be readily seen, this course requires the application of *existing* tools whose theory of operation is covered in subsequent courses.

### 2.4. Iteration 1

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, whereby commands which operate on the standard output and input files can have these file's 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 perform these operations on previously opened files which are specified by number. In addition, there is syntax for files to be entered from the terminal 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 was greatly expanded.

### 2.5. Iteration 2

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 all of the C source files and headers in the current directory. These patterns can be arbitrarily complicated; see Bourne [4] for details. The design of these additions involved several components, of which the most important were 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 was spent on regular expressions, which the students do not encounter formally until the Computability course. Once the regular expression notion was understood, the construction of a pattern matcher became an exercise in coding. The students were advised to first implement a single directory pattern expansion routine, which could then be recursively applied to the multiple directory case. Thus, the students were 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 we used is the fact that new features must be integrated into the 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 were exposed to the issues of software maintenance [14] in a most practical fashion.

### 2.6. Iteration 3

In the third iteration, there were two essentially disjoint additions to the command interpreter. These were 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 were 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. Iteration 4

The fourth and final addition to the command interpreter were the three types of quotation marks employed by the UNIX Shell, ', `, and " [4]. This addition was 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 ` mark, 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.

Particular attention was paid here 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`".

### 2.8. Lessons Learned

Personally, we have found that our mistakes are among the most valuable learning experiences we've had, and know this is not uncommon. Accordingly, we required the students to submit a "Lessons Learned" document, summarizing the positive and negative experiences they had had with tools and methodologies. In order that they understand what such a document was to contain, an example was given detailing our own problems in constructing the command interpreter.

## 3. Discussion

Several other points deserve mention before comparing our approach with others in an abstract sense.

First, 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 assignment of environment on which class time would have been ill-spent.

Second, the choice of an existing software system had several positive effects.

1. The students were not forced to carry out the complete design process before they were capable. Their design process consisted of analyzing a new feature in the context of their existing software, designing appropriate data structures and algorithms, and implementing the feature.
2. The command interpreter they were constructing is completely documented [4], and like any such command interpreter, interactive, a programming language, an an interface to an underlying virtual machine provided by the operating system. In addition, it is an exemplary piece of software design.
3. 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. In fact, several groups of students corrected the instructor on interpreter details based on their independent experiments (sometimes success can be embarrassing!).

Third, the instructor completed all assignments, and generally made the results available on-line. This served both to provide feedback on the complexity of the assignments, and to give the instructor the insight and mastery of detail necessary to aid the student in all phases of the design process.

Fourth, grading of all programming assignments previous to the project was done on the basis of 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 on the basis of 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, rather than the effort, methodology, or document formatting skill. This is as it should be.

### 3.1. Our approach

Having presented details of our course in the previous section, here we will try to abstract the principal accomplishments. Aside from the trivial achievements of introducing the students to C and the UNIX programming environment, we feel that the course structure has several strong points:

- The student develops a non-trivial *toolkit*, consisting of both techniques and developed skills with software tools which can be used later in the Columbia curriculum. The proof of the approach is in, for example, the re-worked command interpreter syntax which students have used in designing database query languages in other course offerings.
- The focus on one significant project brings out the point of software engineering, which is only apparent with scale and re-use.
- 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 was illustrated by the example of *<name, value>* string values.

- 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 [5] are postponed until the student has encountered them, and can appreciate the solutions.

Private discussions with other faculty reinforce our belief that the toolkit approach has value in our setting; a discussion of lexical analysis and parsing certainly makes more sense when the student has already encountered these things; with some practical exposure, the theory not only becomes more accessible, but more relevant.

### 3.2. Related Work

Ideally, software engineering would be the focus of the curriculum, as it was with the Wang Institute [2, 12] Master of Software Engineering (MSE) program. Then, a certain level of pre- and co-requisites and background could be expected, and sufficient attention paid to all aspects of the software lifecycle. Such graduate courses [6, 16] presumably have the advantage of prior exposure to CS on the part of the students, and thus can direct more energy towards software engineering, and less towards the "glue" connecting software design to other areas of CS. An interesting observation is that in the 1987 article on the Toronto course, 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 [7] course was more balanced than ours, both in the selection of students (ranging from freshmen to graduate students) and in the coverage of different portions of the life cycle. Her article actually provides a course outline, with interjected textual comments. The feedback from the students was similar to our own; that is, the course required too much work for the number of credits. Her group size of 5, versus our 3.

Our course offering agrees quite well with the survey results gathered by Leventhal and Mynatt [11] 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 minimal (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 [3] setting is quite similar to ours, although their course offering appears to be slightly later in the West Point curriculum than ours is in Columbia's. It is interesting to note that they took the approach of using many smaller exercises to teach software engineering principles. This contrasts somewhat with our approach of using a single large project, partitioned into development stages. Our focus was on developing a toolbox the student could carry with them into industry or further academic forays; while Bentley and Dallen make a strong argument for their use of AWK [1], we are not sure that an AWK toolkit would be as useful. We are convinced, however, that our approach is integrated well with other portions of our CS curriculum, which it both builds upon and reinforces.

## 4. Conclusions

We believe that the construction of significant software systems is a necessary part of an undergraduate education in CS. There is a transition in our computer science curriculum between introductory and more advanced courses which requires facility with a set of software tools. The SDL course described in this report meets our educational and curricular goals (as stated in the introduction); this approach has proven itself, as our graduates succeed in both academic and industrial settings.

## 5. Acknowledgments

We must thank the students of SDL; previous instructors of the course, Drs. Rodney Farrow and Gerald Leitner; Peter Sweeney, John Ioannidis, Perry Metzger, Lowell Kaufman, Ben Fried, Nick Christopher, Seth Strumph, and Chris Maio; Drs. Steven Feiner and Gerald Q. Maguire, Jr., whose ideas helped develop and refine SDL; and Boeing Aerospace Corporation, AT&T Bell Laboratories, and Bell Communications Research, for their contributions to my practical expertise in software development. AT&T should also be thanked for the donation of many copies of the "UNIX" Issue (July-August, 1978) of the Bell System Technical Journal used in the course.

## 6. References

[1] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, *The AWK Programming Language*, Addison-Wesley, Reading, MA (1988).

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

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

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

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

[6] 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).

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

[8] Brian W. Kernighan and Dennis M. Ritchie, "The M4 Macro Processor," in *Unix Programmer's Manual*, Bell Telephone Laboratories (July, 1977).

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

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

[11] 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).

[12] 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).

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

[14] N. F. Schneidewind, "The State of Software Maintenance," *IEEE Transactions on Software Engineering* SE-13(3), pp. 303-310, Special Section on Software Maintenance (March, 1987).

[15] School of Engineering and Applied Science, *Columbia University Bulletin*, 1986-1987.

[16] 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).