A COURSE ON SOFTWARE ENGINEERING TECHNIQUES

D. L. Parnas
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

INTRODUCTION

This is a report on a course entitled, "Software Engineering Methods", which I have taught to undergraduate students at the Carnegie-Mellon University twice during the 1970-71 academic year. The course is "project oriented" and aims to educate by providing experience in the use of the techniques taught.

WHAT DO I MEAN BY "SOFTWARE ENGINEERING"?

The term "software engineering" is often used to denote the building of commonly used systems programs such as assemblers, compilers and operating systems. In the design of this course I have taken a much broader view. I take the view that programming is taught in our basic courses as a solo activity. Such courses teach programming techniques that are suitable for use by a single person constructing a program which will not be touched by other people. In contrast, I feel that the essential characteristic of a software engineering task is that many people will be involved with the product. Either several people will cooperate in producing it, or it will be used or modified by persons other than the original writer. The course emphasizes procedures which are optional and might be superfluous for solo programming tasks but are important if several people are involved. I shall list those techniques later.

It is certainly possible to complete a multi-person project without those techniques; it is done constantly, but the results are usually unsatisfactory. On the other hand, I believe that often the techniques are useful and appropriate for the construction of a program whose author will be its sole user. A software engineer must be able to communicate precise partial descriptions of the system to others on the project. Because of our limited mental capacity, techniques for communication with others are also used for communication with one's self. I expect the course to result in an improvement of the solo programming skills of the students as well as to prepare them for software engineering projects.

BASIC EDUCATIONAL PHILOSOPHY

This course shares with a hardware course reported previously [1,2] the following basic educational philosophy.

1. It is better to teach methods of problem solving than to teach known solutions to specific problems.
2. It is more important to improve a student's ability to read the literature critically himself than to digest it for him.
3. Students learn better by solving problems themselves than by having problems solved in front of them.
4. It is the role of the university to teach how things should be done, rather than current practice.

Although few would argue with the above "motherhood principles" they have a drastic effect on the course because of the limited time available. There are many useful and well thought out software engineering methods (e.g., syntax analysis methods, list processing algorithms, sorting algorithms, code optimization algorithms, address assignment and subprogram linking) which I treat briefly or ignore in accordance with the above priorities. Many educators will find the content inadequate. Time pressures forced us to make choices in the direction indicated above. Later I mention some specific steps to reduce the conflict.

SUBJECT MATTER

I have often heard it stated that "software engineering" cannot really be taught, that there is no subject matter, that it is an art rather than a science. Although the above educational philosophy reflects some agreement with that view, there follows a list of subject areas which the course emphasizes. In the following, I provide references to other reports rather than reproduce material available elsewhere.

1. Techniques for precisely defining what a piece of software is intended to do [3, 14].
2. Criteria to be used in decomposing software into "modules" or work units [4,5].
3. Criteria to be used in determining the information about each module to be presented to other modules (i.e., interface design) [4,5].
4. Techniques for specifying the functions of modules [6].
5. Techniques for verifying the correctness of specifications.
6. Program organization techniques [7,8,9, 10].
7. Ability to read the software literature, specifically:
    a. familiarity with some of the jargon used in the literature
    b. common assumptions about the structure of systems programs used in the literature

c. paper analysis skills

8. Some issues of language design, e.g., use of syntax, ability to process own text, ease of learning vs. ease of use.

9. Familiarity with the most common system programming problems and tools, e.g., assemblers, compilers, interpreters, libraries, loaders, operating systems.

10. Some "theory of design methodology", e.g., the effect of decision ordering on the final design, hierarchies in program structures, need for constant decision verification [4,10,11], etc.

COURSE ORGANIZATION

The course may be thought of as divided into three phases based upon the type of assignments given to the students. In the first phase the assignments consist of introductory small projects. The students are given definitions of relatively small devices common in software engineering (e.g., a stack, queue or tree structure). For each object some are asked to produce implementations while others are asked to write small programs which use the object. One example of such an object is the stack defined in [6].

In the second phase of the course the class builds a "family" of small systems from a design presented to them by the instructor. The project is a small scale system but larger than the previous projects. The past projects for this course have been a translator for a Markov Algorithm based programming language [12], and a system for the production of KWIC indices. The system is divided into approximately six modules. Each module is given a precise definition and each student builds one module. With 18 students in the class this provides us with three versions of each module. Because of the precise definitions, any correct version of a given module is replaceable by another student's correct version of the same module without changes in any other modules. Thus if all the students do their jobs properly, the "family" consists of 3^6 or 721 working versions. The students working on a given module do not cooperate; the various versions of each module have very different implementations. The ability to produce a system as a team effort with independently replaceable parts is an essential skill for all software engineers.

In the third phase of the course another system is started. In this one the students are given only a rough picture of what the system is intended to do. The class, working as a design committee or system committee, goes through the exercise of squeezing the real intentions of management (in this case the instructor) from the vague description and conversations, producing a more precise form such as is given to them in the first phase. They then go through the exercise of dividing the system into modules, providing precise definitions of the modules, and (if time permits) completing the system as in the first project.

Throughout all three phases the lectures are coordinated with the projects so as to explain to the students what they are doing and why they are doing it. In the introductory projects the main effort is on teaching the students how to use the specifications. Some time is spent on exploring possible implementations with the aid of a programming textbook. The text by Knuth [13] has been found tremendously useful in this regard. During the second phase the project's design is motivated for the students. They are shown how the system's decomposition into modules was arrived at, they are given the reasons for defining the interfaces chosen, and they are shown some alternative formulations together with the relative advantages and disadvantages. An example of the type of comparison which can be made in class is given in [5]. We examine possible implementations of each module, taking care to show several alternatives and show the situations in which each is preferred. In this way I encourage the students to make different implementations and show them the advantages of having several different "plug compatible" implementations. Thus the lectures in the second phase are used to build up a concept of how to design a system. In the third phase the lectures are devoted to the discussions of an actual design and the students are encouraged to apply the conclusions reached in the second phase discussions.

As one compares the course organization with the list of topics to be covered one can see that no attempt has been made to base the course on that list. Instead, the list has been used to motivate the problems and examples used in class. The projects are chosen to either illustrate the most common software components, to provide situations in which the student can profitably make use of the literature to get his own assignments done, or to make him familiar with an important technique. As a result the "factual material" which might be the direct aim of a more conventionally organized course appears as incidental byproducts of the project work of this course. The success of the course in covering the topics that one would like to see covered is dependent very much on the ability of the instructor to select appropriate examples.

RESULTS AND PROBLEMS

To date the results have been encouraging. The problems of the first phase have been completed successfully. The systems of the second phase have always been completed in many versions, though usually we have one version of each module which could not be gotten to work. Instead of our goal of 3^6 working versions the usual result has been about 2^6.* Needless to say, we have not tested all possible combinations, but we have used each module in several combinations. In the third phase we have not had time to proceed to the point of implementation, and I feel that a great deal is lost because of that. One positive note: with one very minor exception, every failure in the second phase project has been traceable to an error in

*Most recently the figures were: 20 students, 5 modules, 4^5 possible versions, 5 incorrect versions (15 correct ones), and 192 apparently correct combinations.

programming a single module rather than an error in the specifications or "system design". From a managerial (or grading) point of view we find that extremely helpful and the next best thing to getting all modules working correctly. In previous project courses [1,2] as well as in current industrial practice, project failure is extremely hard to trace to its source. One usually finds a number of misunderstandings about what each member of the team was supposed to do. Generally the vagueness of the original natural language documents make it impossible to place the blame squarely and often means that a single error is shared by many modules. I feel that the reduction of the "fault placing" problem is a basic verification of the validity of the methods being taught by the course.

CONCLUSIONS

It is tempting to conclude from the above that the course should be taught in a two semester sequence. Certainly there is much more about software that I would like to get across. However, a factor not yet mentioned leads me to postpone that conclusion.

In each semester the class has been relatively easily divided into two groups. One group, which I would call "experienced" had had some previous contact with the making of systems programs. These students had in most cases completed the basic courses offered more than a year before and had part time or summer jobs with some organization writing systems programs. The other group, which I would call "naive" had just completed the more basic courses and had not had any systems programming experience. The first surprise was that the experienced group did not outperform the naive group. The methods being taught were sufficiently different from current practice that both groups were on an equal footing.

Experience did show that it was extremely difficult to give lectures and problems that were suitable for both groups. There were many questions raised by the experienced group which did not seem relevant to the naive group. They were questions relating to current practice with which the naive group was not familiar. On the other hand, there were concerns of the naive group (generally specific programming problems) which were not relevant to the experienced group. Thus, I had a tendency to bore one group or confuse the other.

Economic conditions permitting, it would seem best to offer two versions of the course; one oriented to students with systems programming experience, the other oriented to the more naive student. In fact, for really experienced students (for example, students in an in house training project for a software company or computer manufacturer) a single project coupled with a well planned series of lectures could accomplish much of the course's purpose in two-three weeks intensive or four-six weeks of part time effort. The course for the naive students would then be able to tackle some of the problems in a more orderly manner and could conceivably cover the desired material in a semester. Under those circumstances we might try to devote some sections of the course to providing students with an experience of "the way things are". Without such a section there is a danger that a student, trained in the way outlined above, will become convinced that the approach is a naive one because it does not deal with many of the problems which exist in current systems. Some time in the course should be devoted to a discussion in depth of those problems so that the student can be prepared to differentiate between problems which are the result of a failure to use the methods taught and problems which are unavoidable or intrinsic.

ACKNOWLEDGMENT

I am greatly indebted to Professor Alan J. Perlis, whose constant encouragement contributed greatly to the course's development. I am also grateful to the first students who suffered through early versions of the course.

APPENDIX

The above paper gives little indication of the content of the course because much of it has been published in the references and it did not seem appropriate to the author to duplicate the material. However, the following sample examination problems may provide some indication of the "flavor" of the course.

INTRODUCTION

In the following module all function values and parameters are integers except where stated otherwise. In the interest of brevity we shall not state this repeatedly. For some values the values are not predicted by the definition. They are chosen arbitrarily by the system. This is done because the user should not make use of any regularity which might exist in the values assigned. The necessary relations between the values of those functions and the values of other functions are stated explicitly. Such incompletely defined functions are noted with an *. The user may store the values of those functions and use them to avoid repeated nested function calls.

Note: fa = father, ls = leftson, rs = rightson,
sls = set ls, srs = set rs, sva = set val,
val = value, del = delete, els = exists ls,
ers = exists rs, und. = undefined.

```
Function spslft
   possible values:  integer
   parameters:  none
   initial values:  p2
   effect:

Function exists
   possible values:  true, false
   parameters:  integer i
   initial values:  exists(0) = true;
                    exists(1:p1) = false;
                    all others und.
   effect:  call EC1 if i < 0 or i > p1
```

```
*Function fa
   possible values:  integer
   parameters:  integer i
   initial values:  fa(0) = 0; all others und.
   effect:  call EC2 if i < 0 or i > pl
            call EC3 if 'exists'(i) = false

Function valdefd
   possible values:  true, false
   parameters:  integer i
   initial values:  valdefd(0) = false;
                    all others und.
   effect:  call EC4 if i < 0 or i > pl
            call EC5 if 'exists'(i) = false

Function val
   possible values:  integer
   parameters:  integer i
   initial values:  und.
   effect:  call EC6 if i < 0 or i > pl
            call EC7 if 'exists'(i) = false
            call EC8 if 'valdefd'(i) = false

Function els
   possible values:  true, false
   parameters:  integer i
   initial values:  els(0) = false; all others und.
   effect:  call EC9 if i < 0 or i > pl
            call EC10 if 'exists'(i) = false

Function ers
   possible values:  true, false
   parameters:  integer i
   initial values:  ers(0) = false; all others und.
   effect:  call EC11 if i < 0 or i > pl
            call EC12 if 'exists'(i) = false

*Function ls
   possible values:  integer
   parameters:  integer i
   initial values: und.
   effect:  call EC13 if i < 0 or i > pl
            call EC14 if 'exists'(i) = false
            call EC15 if 'els'(i) = false

*Function rs
   possible values:  integer
   parameters:  integer i
   initial values:  und.
   effect:  call EC16 if i < 0 or i > pl
            call EC17 if 'exists'(i) = false
            call EC18 if 'ers'(i) = false

Function sval
   possible values:  none
   parameters:  integer i,v
   initial values:  not applicable
   effect:  call EC19 if i < 0 or i > pl
            call EC20 if 'exists'(i) = false
            call EC21 if 'valdefd'(i) = true
            val(i) = v
            valdefd(i) = true

Function cval
   possible values:  none
   parameters:  integer i,v
   initial values:  not applicable
   effect:  call EC22 if i < 0 or i > pl
            call EC23 if 'exists'(i) = false
            call EC24 if 'valdefd'(i) = false
            val(i) = v

Function del
   possible values:  none
   parameters:  integer i
   initial values:  not applicable
   effect:  call EC25 if i ≤ 0 or i > pl
            call EC26 if 'exists'(i) = false
            call EC27 if 'els'(i) or 'ers'(i)=true
            fa(i) is und.
            val(i) is und.
            ers(i) is und.
            els(i) is und.
            valdefd(i) is und.
            exists(i) = false
            if i = 'ls'('fa'(i)) then (
               ls('fa'(i)) is und.
               els('fa'(i)) = false)
            if i = 'rs'('fa'(i)) then (
               rs('fa'(i)) is und.
               ers('fa'(i)) = false)
            spslft = 'spslft' + 1

Function sls
   possible values:  none
   parameters:  integer i
   initial values:  not applicable
   effect:  call EC28 if i < 0 or i > pl
            call EC29 if 'exists'(i) = false
            call EC30 if 'els'(i) = true
            call EC31 if 'spslft' = 0
            there exists k such that (
               0 < k ≤ pl
               'exists'(k) = false
               exists(k) = true
               ls(i) = k
               els(i) = true
               els(k) = ers(k) = false
            valdefd(k) = false
               fa(k) = 1 )
            spslft = 'spslft' - 1

Function srs
   possible values:  none
   parameters:  integer i
   initial values:  not applicable
   effect:  call EC32 if i < 0 or i > pl
            call EC33 if 'exists'(i) = false
            call EC34 if 'ers'(i) = true
            call EC35 if 'spslft' = 0
            there exists k such that (
               0 < k ≤ pl
               'exists'(k) = false
               exists(k) = true
               rs(i) = k
            valdefd(k) = false
            els(k)=ers(k) = false
               ers(i) = true
               fa(k) = i )
            spslft = 'spslft' - 1
```

1. Specific questions

   1.1 Can there be two integers i_1 and i_2 such that $ls(i_1) = rs(i_2)$?

   1.2 Give the values of i such that the following ALGOL program will stop.

```
i = some initial value given outside
    the block
```

```
begin
   integer k; k := i;
   L: k := fa(k)
      if k ≠ i then go to L;
end;
```

1.3 Can there be two distinct integers i_1 and i_2 such that $fa(i_1) = fa(i_2)$?

1.4 Consider that the following sequence of calls has been made:

```
sls(0); srs(0); sls(ls(0)); srs(rs(0)); sls(rs(0));
sval(0,0); sva(ls(0),1);
i := ls(ls(0)); sva(i,1); sva(rs(0),3);
sva(rs(rs(0)),4); k = rs(0); sva(ls(k),5);
```

Give the values of the following function calls after the above sequence is executed.

```
val(k);
val(fa(i));
fa(fa(i));
val(fa(fa(i)));
val(rs(rs(fa(i))));
```

Is val(i) = val(ls(ls(0))?

For how many values of i is els(i) true?
For how many values of i is ers(i) true?

1.5 Is there a "longest" expression that will not call an error call? If so, give it. If not, state why not.

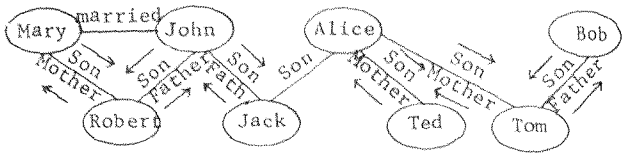
1.6 What sequence of function calls will guarantee that ls(rs(0)) is equal to 5?

2. This module was intended for use in an information retrieval project where a file of publications is stored and each is classified by a binary code consisting of 48 binits (either 0 or 1). Associated with each document is an accession number which tells where to find it on the shelves. The conception of the system is that the user answers up to 48 questions while istting at an interactive terminal (answering either yes or no to each one), then the system prints up an accession number. The number of questions you are asked may well vary with the answers you give. Some answers will suddenly result in a "WE AIN'T GOT NONE BUDDY" reply.

   Show how to use the modules in this application.

3. Can you use this module in a simple direct way to keep the marriage and birth records of a city, i.e., to store information showing the father and mother of each person, the wife of each person, the sons and daughters of each person?

   In other words, how would you store the following information structure with this module? If you think that this is not a good module for this application, say why.



Optional Question (extra credit possible)

For one of the two applications suggested for this module (preferably one where the module is easily applied) show the decomposition into modules that you would use. Describe each module briefly indicating the special piece of knowledge it would hold and the way that it would be used by the rest of the system.

References

[1] Strauss, J. C., D. L. Parnas, R. Snelsire, Y. Wallach, A Design-Emphasis Problem Solving Experience, Electrical Engineering Department, Carnegie Institute of Technology, Pittsburgh, Pa.

[2] Parnas, D. L., "On the Use of the Computer in Engineering Education Without a Programming Prerequisite", Journal of Engineering Education, April 1966.

[3] Parnas, D. L., "Sample Man Machine Interface Specification - A Graphics Based Line Editor", presented at NATO Advanced Study Institute on Man Machine Interaction Using Graphics, held at the IMD, University of Erlangen, Erlangen, West Germany, April 1971. To be published in the Proceedings of that meeting.

[4] Parnas, D. L., "Information Distribution Aspects of Design Methodology", Proceedings of IFIP Congress 1971.

[5] Parnas, D. L., On the Criteria to be Used in Decomposing Systems into Modules", Carnegie-Mellon University Technical Report, to appear in the Communications of the ACM (Programming Techniques Department).

[6] Parnas, D. L., "A Technique for Software Module Specification with Examples", to appear in the Communications of the ACM (Programming Techniques Department).

[7] Wulf, W. A., "Programming Without the Goto", Proceedings of the IFIP Congress 1971.

[8] Dijkstra, E. W., Notes on Structured Programming, Report of the Technical University of Eindhoven.

[9] Dijkstra, E. W., "Structured Programming", in Software Engineering Techniques, edited by Buxton and Randell (available from NATO, Brussels).

[10] Dijkstra, E. W., "A Constructive Approach to the Problem of Program Correctness", BIT 8 (1968), 174-186.

[11] Parnas, D. L., "The Application of Modelling

to System Development and Design" (invited paper), International Computing Symposium, ACM European Chapters, May 1970, vol. IV, 137-147.

[12] Galler, B. and Perlis, A. J., A View of Programming Languages, Addison Wesley, 1970.

[13] Knuth, D. L., The Art of Computer Programming, Volume 1 - Fundamental Algorithms, Addison Wesley.

[14] Parnas, D. L., "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System", Proceedings of the 1969 National ACM Conference, 379-386.