SOME CONCLUSIONS FROM AN EXPERIMENT IN SOFTWARE ENGINEERING TECHNIQUES

David L. Parnas
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa.

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research.

ABSTRACT

This paper describes the outcome of a small experiment to test the validity of some proposed software engineering techniques. The experiments showed that it was possible to combine the work of many programmers to produce systems which could exist in many versions. Major changes in the system's implementation could be made by replacing well defined small subsets of the system. The experiment seems to support (1) the validity of the techniques being tested and (2) some new conclusions about project management and the focussing of effort in software projects.

In two earlier reports [1,2] we have suggested some techniques to be used in producing software with many programmers. The techniques were especially suitable for software which would exist in many versions due to modifications in methods or applications. These techniques have been taught in an undergraduate course [3] and used in an experimental project in that course. The purpose of this report is to describe the results that have been obtained and to discuss some conclusions which we have reached. The experiment was completely uncontrolled, the programmers generally inexperienced and poor, and the programming system used was not designed for the task. The numerical data presented below have no real value. We include them primarily as an illustration of the type of result that can be obtained by use of the techniques described in the earlier reports. We consider these results a drastic improvement over the state of the art. Major changes in a system can be confined to well-defined, small, subsystems. No intellectual effort is required in the final assembly or "integration" phase.

The Project

The class was asked to produce the KWIC index system described in [2]. The project was divided into six modules, but two were combined because they were clearly simpler than the remaining four.* For each of the five assignments we specified four distinct types of implementation. Each student was given one of those to program. Had the experiment been a complete success, any combination of one version of each assignment would have run correctly; we would have had 4^5 working versions (five independent selections from sets of four elements). In addition, each student was assigned to write a program which would "checkout" some module other than his own. Because of the billing policies of our University Computing Center, the programs were to be written and run in WATFIV - a version of FORTRAN. All the defined functions were to be made available as either subprograms or FORTRAN functions.

*See Appendix 1 for a brief description

any measure,two of the poorest students in the class.)

4. This program was clearly incorrect, but still did not violate the restrictions specified for the modules which it called. Thus combinations involving this program would run but would produce incorrect output. It produced the same incorrect output in every combination tested. The program was "completed" by the student well past the due date and the "checker" was not able to do his job.

5. This program simply failed to terminate in any case. The error was found by the checker.

TABLE II

Combination Tested					Execution Time (sec.) (includes compilation of 6-8 sec.)
1A	2B	3B	4B	5A	37.26
1A	2D	3D	4B	5A	11.42
1A	2D	3A	4C	5A	10.87
1B	2E	3A	4C	5A	10.31
1A	2E	3A	4B	5D	8.53
1B	2A	3E	4C	5B	21.79
1A	2A	3B	4B	5B	302.99
1A	2A	3B	4B	5A	50.16
1A	2A	3B	4C	5A	36.69
1A	2A	3D	4C	5A	11.07
1A	2B	3D	4C	5A	10.99
1A	2A	3B	4E	5A	43.30
1A	2D	3B	4E	5A	43.61
1A	2D	3B	4E	5D	19.17
1A	2E	3B	4E	5D	19.16
1A	2E	3B	4C	5D	28.48
1A	2B	3B	4C	5D	27.23
1A	2B	3D	4C	5D	8.43
1A	2B	3D	4C	5B	76.34
1A	2B	3D	4B	5B	113.32
1A	2B	3B	4C	5B	238.88
1A	2B	3E	4C	5D	10.06

Further Experimentation

1. When an earlier version of this note was circulated privately early this year, Mr. Thibault of IRIA, Rocquencourt, France studied the data and suggested trying the combination 1B,2B 3D,4E and 5D which he believed would be significantly faster than any of those tested. [4]. It ran in 4.4 seconds.

2. We have just repeated the whole experiment with a somewhat larger class. The results were essentially the same. We estimate that the family of programs has 1100 members, more than 40 of these were tested. Performance improves somewhat, ranging between 3 and 13 seconds. The only interesting distinction between the two experiments was that the instructor (project leader) changed from intensely interested to bored and unconcerned with no noticeable effect. We also eliminated the problem with storage limitations mentioned above.

Conclusions

1. We cannot avoid stating our conclusion that the experiment has revealed some validity in the comments of our earlier papers [2,3]. Clearly one purpose of this paper is to draw your attention to those earlier ones.

2. Our most significant new conclusion comes in the area sometimes called "project management". Recent papers (e.g. [5]) have suggested that the project manager must devote a significant part of its best manpower to the "integration phase". In our experiment the "integration phase", while not mechanised, was so simple that it could have been mechanised. Even in the few cases where errors did occur, the system had been structured in such a way that diagnostic messages automatically indicated the module making the error. We had no need for anyone who had a thorough knowledge of the whole system. Our experience indeed suggests that the integration phase is a very poor place to invest one's manpower. The limited capacity of our minds makes us more efficient when our job depends on a relatively small amount of knowledge. Moreover, if we plan our project management around a large "integration phase", we will have to invest that manpower again whenever we change some part of the system.

Our experiment suggests that manpower can be much more profitably invested in the "pre-programming" or "design" phase. The success of our project depended largely upon the precisely written module specifications described in [1]. The "cost" or intellectual effort required to produce one of these module specifications was comparable to the cost of producing an implementation of the module. Such predesign work therefore appears to many as unjustifiable overhead. When we amortize this cost over the number of versions of the system which are finally built, and consider the savings realized in the final "integration" phase, it appears to us that the overhead is well justified.

Efforts in the industry to invest heavily in a "pre-design" or "concept" phase have often proven fruitless because the ourcome was a set of natural language documents which were so general that they provided almost no decisions to guide the development groups. When this predesign phase produces precise module specifications the payoff is much more significant.

Additional amortisation of the "pre-design" effort can occur when the modules or their specifications are used (either unchanged or slightly modified) in a later project.

3. Another important conclusion lies in the area of documentation. Several firms have invested heavily in formalized documentation standards intending to make all information easily available to everyone on the project. Our experiment suggests that the effort in these projects can be focussed. Precise documentation of the external characteristics of each module is essential and should be in a standard notation. Our project had minimal documentation about the internals of the one-man assignments. Industrial practice would require more effort in the area than we put into it, but much less effort than is now common. More significant, the specifications produced in the pre-design phase were the only external documentation required throughout the project. These documents were updated several times as errors were discovered, but no additional descriptive material was needed. This is yet another way that the effort invested in the pre-design phase can be amortized.

4. Our experience demonstrated the importance of careful attention to the possibility of errors in the running program during the "pre-programming" phase. Because of our careful attention to the errors in the design phase, errors which did occur when the systems were assembled

were quickly traced to their source and meaningful diagnostic information was produced with almost no effort on the programmer's part. A paper reporting what we have learned in this area is in preparation.

5. Our experience has indicated the great value of independant module tests (by persons other than the module author) before integration. In an earlier effort of this sort we required each programmer to test his own module before integration. In the two experiments which we discuss here, we required an additional person to test the module against the formal specifications (another use of our predesign efforts). Our success rate increased drastically and there were apparently two reasons:

(1) Sloppy programmers do sloppy tests.

(2) The specifications, although precise, can be misinterpreted by human programmers. A misinterpretation by the programmer which resulted in an error in his module often results in a corresponding error in his tests. An independently written test was unlikely to share the same misconceptions.

We are well aware that, as E.W. Dijkstra has put it [6],"Program testing can be used to show the presence of bugs, but never to show their absence." Showing the presence of bugs however is a very valuable service.

We eagerly await the day that professional programmers habitually produce programs which are written so that they can be carefully proven to be error free. In the meantime we suggest that effort invested in independent pre-integration testing is well worthwhile.

Our experience also suggests that both the hierarchical structure which can be found in the system [2] and the abstract nature of the modules themselves greatly ease the building of the "scaffolding" required for independent module tests. To test a given module one needs simulate only those modules immediately below it in the system hierarchy. Further, the nature of the modules means that many of them can be directly simulated by arrays for testing purposes.

NON-CONCLUSIONS

The reader of this paper and the references might be led to some conclusions which those closer to the project would not draw. We mention them here to avoid midleading our readers.

1. The KWIC index structure given in [2] is the best known. FALSE!

Our experiment showed us a number of faults in the design which we are now trying to remedy.

2. Writing a system in a higher level language such as FORTRAN helps to produce a better structured system. FALSE (or at least not supported by our experiment) ! We used FORTRAN because of the billing and priority policies of our computation center. Use of the language actively interfered with some of our efforts imposing quite unnecessary restrictions on what we did. This was especially apparent in the area of error handling. The secret of our success seems to lie in the module specifications which were language independent.

3. D.L. Parnas is a good project manager. FALSE! Experience has shown him to be absent minded, inattentive to details, unaware of the passage of time, forgetful, etc.,etc. The project succeeded in spite of his being in this role.

4. The students in the course were good professional programmers. FALSE! Most of the programs written were horrid by any professional standards. The experiment succeeded in spite of the programmers as well. (There were a few good programs but they were notable exceptions).

5. Communication between modules should always be by subroutine call as it was in the sample system. FALSE! If one divides a system into modules according to the criteria given in [2] the use of subroutine calls imposes a terrible overhead.

Two more non-conclusions

Several writers (e.g. Dennis [7]) have suggested that a hardware supported virtual memory and a language with the ability to pass complex data structures are necessary conditions for well structured or "modular" programs. Neither of these "necessary conditions" were met in the experimental system we are discussing.

We did not need the ability to pass data structures as parameters (all parameters were integers) between modules because of the nature of the way that our system was divided into modules. Data structures were always operated upon within a single module. We suggest that there is often a false identification of the modular structure seen at design time with characteristics of a program when if is running. This however is a very complex issue and we cannot discuss it further here.

Our programs were written in FORTRAN and could have run either with or without the virtual memory mechanism. This however is begging the question because we built a small system where overlays were not necessary. Memory assignment could be done at compile time or assembly time and would be fixed while the program was running. It is definitely true that memory assignments are data which should not be shared between modules but should be hidden from all but one [8]. This allows (in fact requires) programs to be written for a virtual memory. However, the implementation of the one virtual memory module can be done in many ways (hardware mapping, run time software, or assembly time software.) The choice between these implementations is determined by performance considerations not by "modularity" considerations. Thus we can agree with the virtual memory recommendation only if it is stated more carefully indicating that the necessary condition is that memory allocation considerations be hidden from all but one "module". As a historical note we might mention, that one well-structured system, the T.H.E. operating system,(which made heavy use of the virtual memory concept) was implemented without mapping hardware using the run-time software option mentioned earlier.

Final Conclusions

We believe that the small scale experiment described above has provided us with some valuable insights into methods of software production. We recognize the danger of applying small scale results to larger scale projects. We hope however that some organization with the facilities for carrying out larger scale projects will cautiously attempt to apply these results to larger scale projects so that we may refine them further.

REFERENCES

[1] Parnas, D. L., "A Technique for Software Module Specification with Examples," Communications of the ACM (Programming Techniques Department). May 1972

[2] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," to appear in Communications of the ACM (Programming Techniques Department).

[3] Parnas, D. L. "A Course on Software Engineering", Proceedings of the SIGCSE Second Technical Symposium, March 1972.

[4] Depeyrot, M., Private conversations.

[5] Smith, Don, "An Organisation for Successful Project Management," Proceedings of the 1972 SJCC, (p. 129).

[6] Dijkstra, E. W., "Structured Programming," Report on a conference on Software Engineering Techniques held in Garmish.

[7] Dennis, Jack B., "Modularity," Course notes form an advanced study institute held at Technical University of Munich, February 1972.

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

APPENDIX I

A Brief Description of the System(s) Built in the Experiment

This appendix is intended for those who have not yet read [2].

The system being built was intended to read in a set of titles and produce an alphabetized listing of all circular shifts of those titles (a KWIC index).

The six modules were:

1. Input - The only module which knew the input format. Programs in this module read the input but called other modules to actually store the data.

2. Output - The only module to know the output format. This program took the information to be printed from other modules, but selected the format of the information on paper.

3. Line-Holder - The only module to know how the titles were stored in memory. The module offered programs which both stored and retrieved the information from memory.

4. Circular-Shifter - The only module to know how the circular shifts were represented in memory. Some versions actually stored all shifts explicitly, others stored only relatively small directory tables.

5. Symbol Table - This module was hidden within some versions of line holder. Programs calling line holder were unaware of the existence of symbol table.

6. Alphabetizer - The only module to know the sorting method which was used. Some versions did all sorting initially, others sorted only as needed.

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Computer Science Department Carnegie-Mellon university Schenley Park Pittsburgh, Pa. 15213	UNCLASSIFIED
	2b. GROUP

3. REPORT TITLE

SOME CONCLUSIONS FROM AN EXPERIMENT IN SOFTWARE ENGINEERING TECHNIQUES

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Final

5. AUTHOR(S) (First name, middle initial, last name)

David L. Parnas

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June, 1972	14	8

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
F44620-70-C-0107	
b. PROJECT NO. 9769	
c. 61102F	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)
d. 681304	

10. DISTRIBUTION STATEMENT

A. Approved for public release; distribution unlimited

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
TECH OTHER	Air Force Office of Scientific Rsch (NM) 1400 Wilson Blvd. Arlington, Virginia 22209

13. ABSTRACT

This paper describes the outcome of a small experiment to test the validity of some proposed software engineering techniques. The experiments showed that it was possible to combine the work of many programmers to produce systems which could exist in many versions. Major changes in the system's implementation could be made by replacing well defined small subsets of the system. The experiment seems to support (1) the validity of the techniques being tested and (2) some new conclusions about project management and the focussing of effort in software projects.